CHALMERS
UNIVERSITY OF TECHNOLOGY

APRNet: Convolutional neural networks for a content-adaptive particle representation of images

Master's thesis in Engineering Mathematics and Computational Science

Joel Jonsson

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's Thesis 2019

APRNet: Convolutional neural networks for a content-adaptive particle representation of images

JOEL JONSSON

Department of Electrical Engineering
Division of Signal processing and Biomedical engineering
Chalmers University of Technology
Gothenburg, Sweden 2019

APRNet: Convolutional neural networks for a content-adaptive particle representation of images
JOEL JONSSON

Supervisors:
Dr. Bevan L. Cheeseman
Prof. Dr. Ivo F. Sbalzarini
Chair of Scientific Computing for Systems Biology
Faculty of Computer Science, TU Dresden & Center for Systems Biology Dresden,
Max Planck Institute of Molecular Cell Biology and Genetics

Examiner:
Prof. Dr. Fredrik Kahl
Department of Elecrtical Engineering
Chalmers University of Technology

Master's Thesis 2019:NN
Department of Electrical Engineering
Division of Signal processing and Biomedical engineering
Computer Vision Group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Image of a cat and a dog, represented by particles placed according to the information content.

Typeset in LaTeX
Gothenburg, Sweden 2019

APRNet: Convolutional neural networks for a content-adaptive particle representation of images
JOEL JONSSON
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Convolutional neural networks have shown outstanding performance in a wide range of image processing and computer vision tasks. The processing of biological fluorescence microscopy images is no exception. However, practical applications are in many cases hampered by the size of the datasets, which typically take the form of 3D stacks of high-resolution pixel images sampled through time. To reduce the memory and computational resources required to store and process such data, the Adaptive Particle Representation (APR) has been proposed as a replacement for the pixel representation. By replacing pixels with particles positioned according to the contents of the image, the APR enables orders of magnitude reductions in the computational and memory cost of storing and processing 3D microscopy images.

In this work, first steps are taken toward a full extension of convolutional neural networks to the APR. This is achieved by designing and implementing the fundamental components of such networks for 2D image classification as operations on the APR. These components are built into custom modules in the popular deep learning framework PyTorch, allowing seamless design and application of the resulting network type, which we call APRNet. The potential of APRNets is demonstrated on image classification tasks using both synthetic and natural images. It is shown that APRNets can achieve equal classification performance to pixel networks of similar design, while enabling significant reductions in both memory usage and computational cost. The results also indicate that processing on the APR may provide additional benefits by imposing a regularizing effect on the learning.

Keywords: deep learning, computer vision, CNN, adaptive image representation, APR, microscopy

Acknowledgements

This project has been conducted at the Center for Systems Biology Dresden (CSBD) and the Max Planck Institute for Cell Biology and Genetics (MPI-CBG) in Dresden, Germany. I owe thanks to many people who have helped and supported me, both professionally and personally, during this time.

First, I would like to thank Ivo Sbalzarini for inviting me into the MOSAIC group and making the project possible. My thanks go to the entire MOSAIC group for providing a wonderful workplace, as well as scientific assistance and discussions. I would like to extend a special thank you to Bevan Cheeseman for supervising, and always going the extra mile to discuss and assist with issues in every stage of the project; and to Krzysztof Gonciarz for assisting with the implementation. I would also like to thank Fredrik Kahl for agreeing to examine the project and providing valuable input. Last but not least, thank you to my friends and family for your unwavering support throughout my studies.

Joel Jonsson, Dresden, March 2019

Contents

1 Introduction

1.1 Background

Deep learning methods, and in particular convolutional neural networks, are finding an increasing number of successful applications in the analysis of biological flouresence microscopy data [1, 2, 3, 4]. This data often comes in the form of 3D stacks of pixel images sampled through time. Depending on the technique used and the required resolution, such images can easily reach sizes where even simple visualization tasks exceed the memory and computational capticity of current techniques and hardware [5]. Hence, the application of deep learning methods, which are inherently demanding both in terms of computational power and memory requirements, is either not possible on the full datasets, or requires special pre-processing and algorithms [6].

The Adaptive Particle Representation (APR) [7] has been developed to alleviate computational and memory bottlenecks in the storage and processing of such large images. It does so by adapting the image resolution according to local information content, enabling orders of magnitude reductions in the size of the representation [7]. Unlike standard image compression techniques, once an image is converted to an APR the adaptive representation can be used for processing and visualization tasks, without returning to the original pixel image. Many algorithms do, however, need to be modified to account for the more complex data structures.

Due to the prominence of deep learning methods in image analysis, their adaptation to the APR is both useful and necessary. Firstly, it would enable the use of powerful deep learning approaches in APR-based pipelines. Secondly, with the methods in place the APR could be used as a tool to significantly reduce the computational resource requirements for applications of deep learning to very large images. Lastly, the formulation and evaluation of classical deep learning approaches, such as convolutional neural networks, directly on such a content-aware, multi-resolution image representation constitutes a theoretically interesting problem.

1.2 Aims and objectives

The aim of the project is to develop and implement convolutional neural networks that operate directly on the APR. In order to achieve this, the fundamental building blocks of a convolutional neural network, e.g. the convolution, down- and upsampling operations, must be adapted to the data structure of the APR. Subsequently, the performance of the developed methods should be evaluated on benchmark learning tasks.

1.3 Limitations

Convolutional neural networks constitute a broad class of algorithms, and may be comprised of various building blocks depending on the task at hand. To constrain the required amount of work, this work focuses on networks for image classification.

Furthermore, much of the recent success of convolutional neural networks can be attributed to efficiently implemented algorithms. In particular, the use of graphics processing units (GPUs) to accelerate computation has had a profound impact. While computational performance is a key topic in this work, the limited time span of the project does not allow for a highly optimized implementation, nor the use of GPUs. The expected outcome is to have a working prototype for central processing unit (CPU) computation, serving as a proof of concept.

1.4 Thesis outline

The remainder of the thesis is structured as follows:

Chapter 2 introduces the concepts and theory necessary to understand the work. First, the problem of image classification is briefly formalized and discussed. Subsequently, convolutional neural networks for image classification are described, and the APR is motivated and introduced.

Chapter 3 describes the adaptation of convolutional neural network operations to the APR, as well as their implementation and related work.

Chapter 4 presents the results of empirical studies of the developed networks for image classification using the APR. This is done on benchmark learning tasks using both synthetic and natural images.

Chapter 5 summarizes what has been accomplished and discusses conclusions that can be drawn from the presented work, as well as possible directions for future work.

2

Preliminaries

This chapter introduces the concepts and theory required to understand the main body of the text. The first section briefly introduces and formalizes the problem of image classification, and describes how a classification algorithm can be "trained" in the supervised learning regime. Subsequently, artificial neural networks, which constitute a family of machine learning algorithms often used for classification, are introduced. Particular focus is placed on convolutional neural networks and their building blocks, as well as the processes of building and training such models. Finally, the adaptive particle representation is motivated and the necessary theory behind it is described.

2.1 Image classification

Image classification is the task of assigning a single label, from a fixed set of categories, to an input image. While most such problems can be trivially solved by humans, there is great value in automating the process. Automated image classification finds applications in software for, e.g., facial recognition in security systems [8] and the detection of cancer or other anomalies in medical images [9].

From a probabilistic perspective, an image x and its "true" label y can be viewed as realizations of random variables X and Y, characterized by a joint probability distribution $P_{X,Y}(x, y) := P(X = x, Y = y)$.[1] If the joint distribution is known, the label of an observed image x is characterized the conditional distribution

$$P_Y(y|x) = \frac{P_{X,Y}(x, y)}{P_X(x)} = \frac{P_{X,Y}(x, y)}{\sum_y P_{X,Y}(x, y)}.$$

With this information one could make an educated guess to classify x, for instance by choosing its most probable label

$$\hat{y} = \arg\max_y P_Y(y|x).$$

However, these probability distributions are unknown and intractable. Instead, they can only be approximately inferred from data. In this work we consider the use of artificial neural networks as data-driven discriminative models. That is, models focused on the direct mapping from images to labels. Note that this does not necessarily equate to estimating $P_Y(y|x)$; the probabilistic reasoning above merely

[1]Both X and Y are assumed to be discrete random variables here, but similar arguments can be made in the continuous case using probability densities.

serves as one example of several possible formalizations of the problem. Prior to introducing neural networks in detail we give a brief description of the problem of "training" a generic classifier in the supervised learning regime.

2.1.1 Supervised learning

Consider a generic image classification model that simply defines a function $f(x;\theta)$, where x denotes an input image and θ a set of free parameters, which outputs a score for each category considered. The functional form of f depends on the specific model, and is here assumed to be fixed. Suppose now that we have a set of labeled images $\{(x_i, y_i)\}_{i=1}^N$, which we refer to as *training data*. These data provide examples of outputs of an optimal mapping $y_i = f^*(x_i)$ at the points x_i. For instance, by encoding the observed labels as vectors of class probabilities with probability one for the observed class and zero for all others[2], $f^*(x)$ can intuitively be thought of as a conditional probability distribution $P_Y(y|x)$. The parameters θ can then be tuned so that f is close to f^* at the points x_i where it is known. Here "close" is defined by a functional $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \longrightarrow \mathbb{R}^+$ on the set $\mathcal{Y}$ of possible labels or class scores. The functional $\mathcal{L}$ is typically referred to as the *loss function*, and often takes the form of a divergence or distance metric on $\mathcal{Y}$. Thus, the model f is presented with inputs x_i as well as corresponding supervisory signals $f^*(x_i)$, and is corrected ("taught") to produce similar signals under $\mathcal{L}$.

Of course, in any practical application the objective is to derive a model that can classify new images, outside of the training data set. Hence, the ultimate goal of the learning process is to find the parameters such that f is as close as possible to f^* for all possible inputs. In the probabilistic framework this can be stated as finding

$$\theta^* = \arg\min_\theta E\left[\mathcal{L}(f(x;\theta), y)\right],$$

where

$$E\left[\mathcal{L}(f(x;\theta), y)\right] = \int \mathcal{L}(f(x,\theta), y) dP_{X,Y}(x,y) \tag{2.1}$$

is the expected loss over the joint distribution of images and labels, referred to as the *risk* associated with the model. Again, since this distribution is unknown the expected value can only be estimated, typically by the sample mean over the training data. This is known as the *empirical risk*:

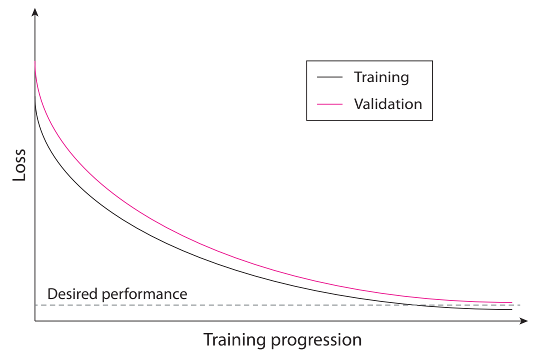
$$E_{train}[\mathcal{L}(f(x,\theta), y)] = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f(x_i,\theta), y_i). \tag{2.2}$$

Training the model is thus stated as the optimization problem of finding the parameters that minimize the empirical risk, which is referred to as empirical risk minimization. However, it is important to remember that (2.2) is only an estimate of (2.1), and low empirical risk does not necessarily imply low risk. In fact, any model with sufficient capacity can achieve near-perfect empirical risk by "memorizing" the training data. That is, the model adapts to the noise in the specific dataset, and most likely provides a poor representation of the underlying structure. This is

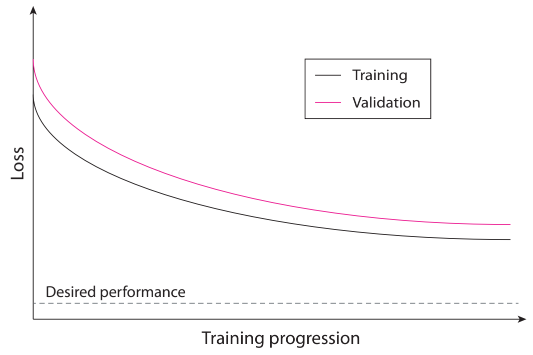
[2]This is often referred to as one-hot encoding.

referred to as *overfitting*. The converse, i.e. when the model capacity is insufficient to capture the underlying structure, is referred to as *underfitting*.

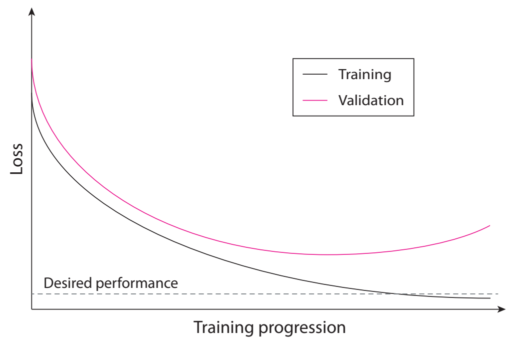
To keep track of how the model performs on data not used in the training process, the available labeled data set is typically divided into three parts: training, validation and test data. The training data are used to update the model parameters, while the generalization performance is estimated using the validation set and tracked throughout the training. If the training loss (2.2) is large, the model is underfitting and its complexity must be increased. Overfitting is instead characterized by small training loss and large, or growing, validation loss. This can be combated by either decreasing the model complexity or applying various *regularization* techniques to improve the generalizability. Some such techniques are described in Section 2.2.4.4. Figure 2.1 shows examples of typical *learning curves*, i.e. the progression of the loss during training with an iterative scheme, when the model is fitting appropriately versus under- and overfitting.



(a) Appropriate fit



(b) Underfitting



(c) Overfitting

Figure 2.1: (a) Example of ideal progression of the generalization error during training. (b) Underfitting is characterized by large training loss. (c) Small training loss and large (or growing) validation loss is an indication of overfitting.

In this way, one can compare different models by their ability to make predictions on the unseen validation data and choose the best-performing model. However, tuning or choosing a model to perform well on the validation data introduces bias to the selection. The performance of the final model is therefore measured on the test data set, which must be held out during both the model selection and training procedures.

2.2 Artificial neural networks

Artificial neural networks (ANNs) constitute a class of machine learning algorithms loosely inspired by the way the human brain processes information. They consist of simple, interconnected processing elements, which are referred to as *neurons*. Each neuron recieves one or more input signals, from which an output signal is computed. A network is formed by passing the output of certain neurons as input to other neurons, typically through weighted connections. The neurons are usually further aggregated into *layers*, such that each layer performs a certain operation. In this terminology, signals are passed to the network through the *input layer*, transformed through a series of *hidden layers* and finally returned by the *output layer*. Hence, at a high level, an ANN simply defines a parameterized function from some input space to some output space.

In the common jargon, the *depth* of a network refers to the number of hidden and output layers, i.e. exluding the input layer. A neural network is said to be *deep* if it comprises multiple hidden layers. Deep neural networks constitute the most fundamental class of algorithms in the field of *deep learning*. Many breakthroughs reported in recent years have used such networks to advance the state of the art in visual recognition as well as the processing of text, speech and audio [10]. The essential idea is that the first processing layers extract simple features from the input data, out of which subsequent layers build increasingly complex and abstract features. In successful applications, these features are highly correlated with the (desired) outputs, allowing the network to make good predictions. For a detailed introduction to deep learning and its applications, the reader is referred to [11].

This section aims to describe *convolutional neural networks*, which constitute a specific class of networks that is at the center of this work. Focus is placed on the underlying mathematics and technical aspects of building and training such networks. First, a brief description of the most fundamental neural network architecture is given, namely the *multilayer perceptron*. This provides a simple introduction that progresses naturally to the more complex convolutional networks.

2.2.1 Multilayer perceptron

The multilayer perceptron (MLP) is a simple neural network architecture consisting of at least one hidden layer and where each layer is *dense*, or *fully connected*. That is, each neuron in a given layer is connected to every neuron in the previous layer, as shown in Figure 2.2. The individual neurons are mathematical functions of their inputs and the corresponding connection weights. More precisely, suppose that a neuron receives n real-valued input signals, denoted by $(x_1, x_2, ..., x_n) = \boldsymbol{x} \in \mathbb{R}^n$, through individually weighted connections with weights $\boldsymbol{\omega} \in \mathbb{R}^n$. The neuron then outputs a value y of the form

$$y = \varphi\left(\sum_{i=1}^{n} \omega_i x_i + b\right) = \varphi\left(\boldsymbol{\omega}^T \boldsymbol{x} + b\right),$$

where φ is referred to as the *activation function*, ω_i denotes the connection weight for input x_i and b is a bias parameter.

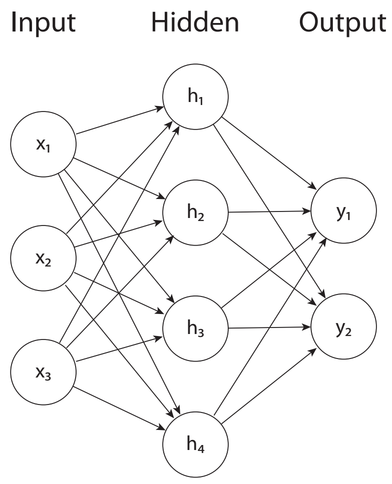


Figure 2.2: Illustration of the connectivity in a multilayer perceptron consisting of 3 input neurons, 4 hidden neurons and 2 output neurons.

The activation functions can be chosen freely, but most training procedures in popular use are restricted to differentiable functions. If the activations are chosen to be linear, the entire MLP becomes a linear function of the inputs. Therefore, nonlinear activation functions are required to enable the network to approximate nonlinear functions. Currently, the most popular choice is the rectifier function

$$f(x) = x^+ = \max(0, x)$$

and variants thereof. It has been shown [12] to yield better performance in deep neural networks than historically popular choices, such as the logistic sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$

or the hyperbolic tangent

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

Multilayer perceptrons are powerful function approximators [13] that find uses in a wide range of regression and classification problems [14, 15, 16]. However, due to the dense connectivity between the layers, the number of parameters can quickly grow to extremely large numbers for wide layers and inputs of very high dimensionality. Furthermore, the dense connectivity means that MLPs learn features at fixed spatial locations. This is often undesirable in image processing and visual recognition tasks. Consider for example the task of classifying images according to the presence of certain objects. The objects are characterized by local relationships between the pixel intensities. A human observer can easily recognize a given object independent of its global location in the image. However, MLPs inherently lack this ability, and would therefore have to be trained to recognize each object at each location in the image. Convolutional neural networks, on the other hand, are specifically designed to respond to local interactions in spatially structured data, and circumvent the problems described above.

2.2.2 Convolutional neural networks

Convolutional neural networks (CNNs) employ connectivity patterns between neurons inspired by the visual cortex in animals [17]. In recent years, they have found groundbreaking success in the processing of images, video, text and audio [10, 18]. Applications of CNNs in image processing cover a wide range of tasks, including image classification [17], semantic segmentation [19], as well as inverse problems such as image denoising [20] and superresolution [21].

Unlike multilayer perceptrons, which consist solely of fully connected layers, CNNs generally employ a number of different layer operations. The most fundamental component of a convolutional neural network is the convolutional layer, which computes its output by means of discrete convolution. Before describing the typical structure of these layers, we give a relatively thorough description of the discrete convolution operation.

2.2.2.1 The discrete convolution operation

Convolution is a mathematical operation that takes two functions as input and produces a new function. For instance, let $f, g : \mathbb{Z} \to \mathbb{C}$. The discrete convolution of f and g at a point $k \in \mathbb{Z}$ is defined as

$$(f * g)[k] := \sum_{n=-\infty}^{\infty} f[n]g[k-n] = \sum_{n=-\infty}^{\infty} f[k-n]g[n]. \tag{2.3}$$

This definition is easily extended to functions on $\mathbb{Z}^d$ by replacing indices with d-tuples and taking the sum over $\mathbb{Z}^d$. If either f or g has finite support, the convolution at a given point is a finite sum. Furthermore, when both f and g are compactly supported functions, their convolution also has compact support.

As a simple one-dimensional example, consider a sequence (or one-dimensional image) $I = (I_k)_{k=0}^{N-1}$, with $I_k \in \mathbb{R}$ for all k. In light of the definition in (2.4), I can be thought of as a function mapping points in $D := \{0, 1, ..., N-1\}$ to $\mathbb{R}$. Similarly define $g = (\omega_{-1}, \omega_0, \omega_1)$.[3] In the context of image processing and CNNs, I is referred to as the input (image) and g as a convolution kernel or filter. The discrete convolution of I and g takes the form

$$(I * g)[k] = \sum_{n=-1}^{1} I[k-n]g[n] = \sum_{n=-1}^{1} I_{k-n}\omega_n,$$

which is well-defined and easily computed for $k = 1, ..., N-2$. However, in many applications it is desirable to produce an output that is structurally identical to the input. This can be achieved by extending the domain of the input, typically by defining $I[m] = 0$ for $m \in \mathbb{Z} \setminus D$. In practice, this amounts to zero-padding the input. In this case, an output sequence of length N can be obtained by defining $I_{-1} = I_N = 0$. With this type of zero-padding, the convolution operation can be thought of as a sliding window operation where the kernel is slid across the input

[3]Note that the indexing only affects the indexing of the output, and not its structure or value. This particular indexing has been chosen to emphasize the center of the kernel, which may ease the intuitive understanding in later parts of the text.

domain, combining values in small neighborhoods to produce an output value at each spatial position of the kernel center. This is illustrated in Figure 2.3. The sizes of the neighborhoods are determined by the spatial extent of the kernel, which is fixed across the domain. This is often referred to as the *receptive field* in the context of CNNs.

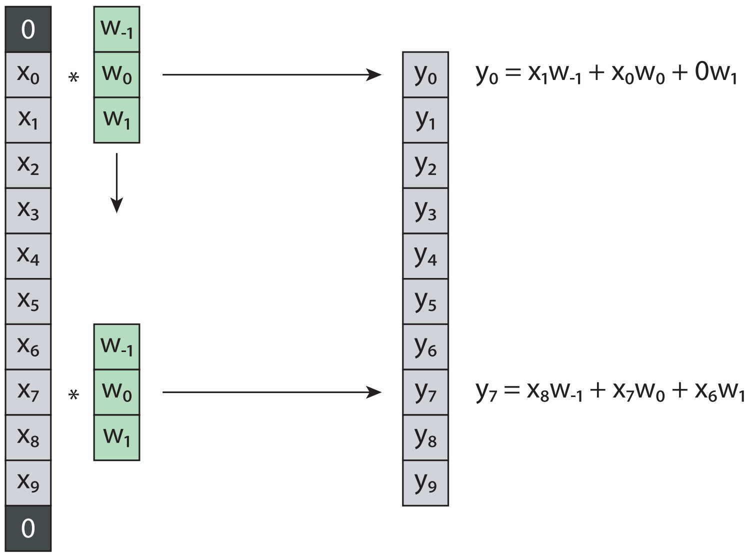


Figure 2.3: Illustration of the discrete convolution as a sliding window operation in one dimension. Zero padding is employed to preserve the spatial structure of the input.

2.2.2.2 Convolutional layers

Now, a convolutional layer essentially consists of a number C of convolution kernels and optionally a set of bias parameters. The input to the layer is convolved with each kernel to produce C outputs, which are often referred to as *feature maps*. Bias parameters, if used, are typically added element-wise to the feature maps, with each feature map receiving a different bias. Furthermore, if the input has several channels, as with color images and the outputs of previous convolutional layers, the kernels span the entire additional dimension. For instance, if the input is an RGB image of size $3 \times W \times H$, where the first dimension represents the color channels, one employs kernels of size $3 \times k \times k$. In this way, the feature map obtained by convolving the input with a kernel corresponds to a 2D image. Hence, the output of the entire layer is a set of C feature maps (images). These are then stacked and treated like a single image with C channels in later convolutional layers. This is illustrated in Figure 2.4.

As an example, suppose that a convolutional layer receives one-dimensional images of size W with C channels. The images are convolved with D kernels of shape $C \times k$, where $k = 2n + 1$, and each kernel has a single associated bias parameter. Further suppose that the input images are padded with n zeroes on both sides. That is, the input is a tensor of shape $C \times (W + 2n)$ with elements $x_{c,i}$. The kernel weights are collected in a tensor of shape $D \times C \times k$ with elements $\omega_{d,c,j}$, with corresponding

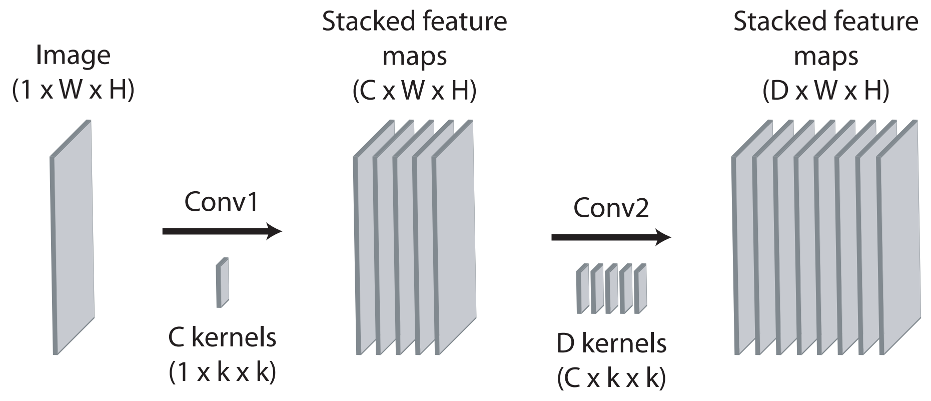


Figure 2.4: Illustration of two consecutive convolutional layers in two spatial dimensions. Discrete convolution operations on stacks of images by constraining the kernels to span the entire additional dimension, so that the kernels are moved only in the spatial dimensions.

bias parameters b_d. Then the output tensor (of shape $D \times W$) is computed according to

$$y_{d,i} = \sum_{c=1}^{C} \sum_{j=1}^{k} x_{c,i+k-j} \omega_{d,c,j} + b_d, \tag{2.4}$$

for $d = 1, ..., D$ and $i = 1, ..., W$. Nonlinear activation functions are then applied to these values before they are passed to subsequent layers.

Despite the seemingly more intricate definition, convolutional layers are in many ways similar to fully connected layers. In fact, fully connected layers can be replicated by using kernels that span the entire input domain. Similarly, a convolutional layer can be obtained from a fully connected layer by restricting certain connection weights to be zero and constraining groups of the remaining connections to have equal weights. Thus, the essential new properties of convolutional layers are sparse connectivity and parameter sharing, both of which are intuitively useful for the processing of certain spatially structured data, such as images. The sparse connectivity forces the layer to focus on local interactions in the input data, while the particular form of parameter sharing induces an equivariance to translations. That is, the local response to a shifted region in the input will be identical and equally shifted. Finally, unlike fully connected layers, convolutional layers accept input of variable size.

2.2.2.3 Downsampling

Downsampling generally refers to the process of subsampling a signal to represent it by fewer samples. Applied to an image, which can be viewed as a sampled signal, the downsampling process reduces its resolution. For example, by discarding every other pixel the resolution is halved or, stated equivalently, the image is downsampled by a factor of two.

In convolutional neural networks, downsampling methods typically take the form of strided sliding window operations. At each position of the window, data points in its range are aggregated via some function to produce a single value. In the machine learning community, this is commonly referred to as a *pooling* operation, and the corresponding layers are called pooling layers. The most common choices of the aggregation function are the max and average operations [11]. Figure 2.5 shows an example of max pooling in two dimensions. When the input comprises multiple channels, each channel is typically pooled independently.

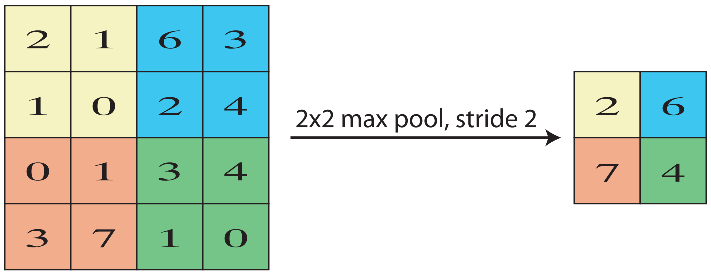


Figure 2.5: Illustration of 2×2 max pooling with stride 2 applied to an input of size 4×4.

Interlaced with convolutional layers in a network, pooling has several effects. By reducing the spatial size of the feature maps, fewer values need to be stored and the operations in subsequent layers require fewer computations. Hence, it reduces the computational cost and memory footprint of the network. Pooling also increases the (indirect) receptive field of subsequent neurons onto previous neurons. As an example, consider the output depicted in Figure 2.5. An operation using all values in the output 2×2 map is indirectly using information from the entire 4×4 map that was input to the pooling operation. In this way, pooling induces a multiplicative increase in the receptive field of subsequent neurons (by a factor equal to the stride of the operation). Lastly, max pooling gives the network some degree of invariance[4] to small translations ("spatial jitter") of the input. While this may be desirable when the task is to simply detect the presence of an object in the image, it may constitute a problem in tasks where the exact location of the object is important.

2.2.3 Network design

The design of a neural network has a profound impact on its capacity and performance [11]. Of course, the choice of architecture depends strongly on the task which is to be solved by the network. For image classification, the general idea is that interlaced convolutional and pooling layers allow the network to extract representative features from images in a hierarchical fashion. The first layer filters typically respond to basic features such as edges and colors, while deeper layers respond to conjunctions of these features, which correspond to increasingly abstract features in the original input [22]. Typical CNNs for image classification therefore consist of two parts: a feature extractor composed of interlaced convolutional and pooling layers, followed by a classifier that predicts class labels based on the extracted features. The classifier, or *head*, often takes the form of a multilayer perceptron.

Older milestone designs such as LeNet-5 (1998) [23] and AlexNet (2012) [24] were composed of relatively few convolutional layers, all of which were followed by a non-

[4]This invariance is not to be confused with the equivariance of the convolution operation.

linear activation function and some also by pooling. AlexNet is often accredited as the trigger for the recent resurgence of commercial interest in deep CNNs for visual recognition, by almost halving the second-best error rate in the ImageNet Large Scale Visual Recognition Challenge [25] (ILSVRC) of 2012. Since then, CNNs have dominated the leaderboards in the ILSVRC and other visual recognition challenges. New network designs that perform well on these benchmark tasks are frequently reported in the literature. Using modern techniques, such as skip connections and residual modules, networks with hundreds or even thousands of layers have been successfully trained [26]. However, this work is focused on the basic operations of CNNs. Therefore, we instead give an overview of a straightforward design that is still in popular use, namely the VGG16 network depicted in Figure 2.6. The VGG team won the ILSVRC-2014 localization challenge and placed second in the classification challenge with this type of architecture [27].

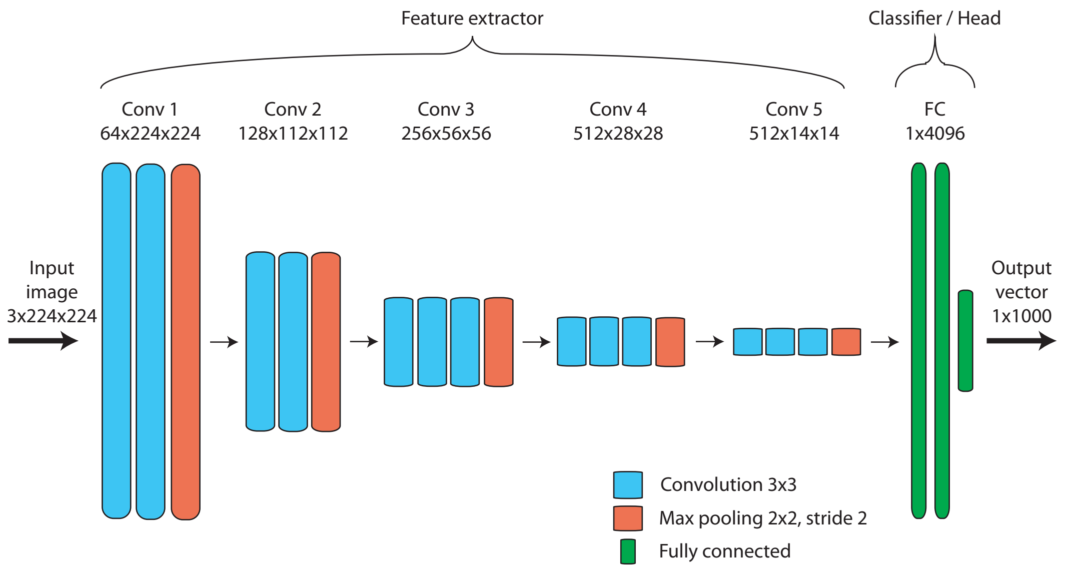


Figure 2.6: Illustration of the VGG16 network architecture for classification of the 1000-class ImageNet dataset. It consists of five convolutional modules and a fully connected head. Each convolutional module is composed of two or three convolutional layers followed by max pooling.

Note that, out of the roughly 138 million parameters in the VGG16 network, only 14.7 million belong to the convolutional layers. The output of the last convolution module, which is of size $512 \times 7 \times 7$, is flattened to a one-dimensional vector with 25088 entries and fully connected to the 4096 neurons in the first layer of the head. This results in a staggering 102.7 million parameters in the first fully connected layer alone. While allowing the head to learn complicated mappings from the learned feature space to the final output, this also makes the network prone to overfitting.

A major drawback of the densely connected head is that it restricts the network to inputs of fixed size, even though the pooling and convolutional layers accept inputs of variable size. There are a number of ways to circumvent this, for instance, by modifying the last max pooling layer to map inputs of variable size to outputs of fixed dimensionality. One such operation is used in the *spatial pyramid pooling* layer [28], which essentially divides its input into a fixed number of bins at different resolution levels and outputs a summary statistic for each bin. A simpler approach is to use *global average pooling*, which computes the spatial average of each feature map, thus transforming a generic input array of shape $C \times W \times H$ into a vector of length C.

2.2.4 Training neural networks

Once the architecture of a neural network has been fixed, it can simply be viewed as a parameterized function. The parameters can then be tuned in various ways to allow the network to perform a certain task. Here we consider the supervised learning approach of empirical risk minimization, as discussed in Section 2.1.1, for the specific task of image classification.

To this end, suppose that we have a CNN $f : X \times \Theta \to Y$ that maps images $x \in X$ to vectors $y \in Y$ of class scores, parameterized by a set of parameters $\theta \in \Theta$. For an input image x, denote by $\hat{y} = f(x; \Theta)$ the predicted class scores and take $\hat{c} = \arg\max \hat{y}$ to be the predicted class. Given a training dataset $D_{train} = \{x_i, c_i\}_{i=1}^N$ of labeled images, with the labels encoded as probability vectors y_i, the network is trained by minimizing the empirical risk

$$E_{train}[\mathcal{L}(f(x, \theta), y)] = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f(x_i; \theta), y_i). \tag{2.5}$$

In this way, training the network is stated as an optimization problem. The following sections describe in detail some of the most commonly applied steps in this process. Firstly, choices of loss functions are discussed. This is followed by a detailed description of the most commonly used parameter updating scheme. Lastly, a few popular regularization techniques are outlined.

2.2.4.1 Loss functions

In order to measure the performance of the CNN, it is logical to consider the *classification accuracy*, that is, the percentage of correct predictions. This is easily computed for a given dataset $D = \{x_i, c_i\}_{i=1}^N$ of images x_i with labels c_i through

$$\text{accuracy} = \frac{1}{N} \sum_{i=1}^{N} 1_{\{\hat{c}_i = c_i\}}.$$

The loss function most closely related to this performance measure is the 0-1 loss

$$\mathcal{L}(\hat{c}, c) = 1_{\hat{c} \neq c} = \begin{cases} 1 \text{ if } \hat{c} \neq c \\ 0 \text{ otherwise} \end{cases}.$$

Substituted into (2.5), it is easily seen that this results in $E_{train} = 1 - \text{accuracy}$. However, the 0-1 loss is inappropriate for direct optimization. By only considering the maximum class score, information about the confidence of the prediction is lost. Furthermore, gradient based optimization techniques cannot be used as its gradient is either 0 or undefined everywhere. For these reasons, it is typically not used in practice. Instead, surrogate loss functions can be used. A common choice is the *cross entropy* loss, which in the specific case considered here takes the form

$$\mathcal{L}(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i). \tag{2.6}$$

Here y_i is 1 if class i is the true label and 0 otherwise, while $\hat{y}_i$ is the predicted probability for class i. These probabilities are obtained by normalizing the output class scores into a probability vector. That is, a vector with entries in $[0, 1]$ that sum to unity.

Since only the term corresponding to the true class is nonzero in (2.6), the loss depends only on the predicted probability of the true class. The dependence is shown in Figure 2.7. It is seen that heavy penalties are given to very low predicted probabilities. As the predicted probability increases, the loss transitions into a gentle descent toward zero for the correct value of one.

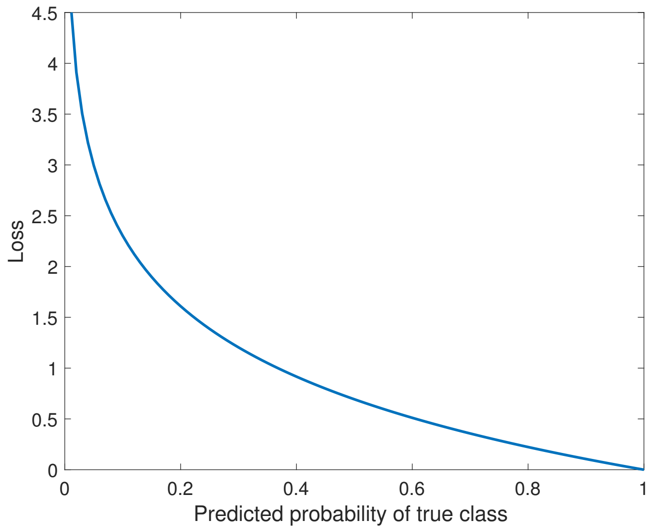


Figure 2.7: The cross entropy loss' dependence on the predicted probability of the true class.

It is intuitively clear that minimizing the cross entropy loss corresponds to maximizing the correct class scores in relation to the faulty ones. Therefore, the classification accuracy is indirectly maximized. In fact, when optimizing with respect to the cross-entropy loss, the test accuracy often continues to increase long after the 0-1 loss over the training set has reached zero [11]. This is because the smoothness of the cross entropy loss allows the robustness of the classifier to continue to increase by further separating the classes.

2.2.4.2 Stochastic gradient descent

With the network architecture $f(x; \theta)$ and loss function $\mathcal{L}$ fixed, training the network is stated as the optimization problem of finding

$$\theta^* = \arg\min_{\theta \in \Theta} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f(x_i, \theta), y_i). \tag{2.7}$$

However, recall that the goal of the training is to minimize the expected loss over the unknown underlying data distribution. Neural network training therefore differs from traditional optimization in that the objective function is unknown, and only estimates of it can be evaluated. Combined with the risk of overfitting, this means that it is typically not desirable to find a global minimum of the empirical risk. It is therefore common to apply iterative optimization methods, while tracking the generalization loss in regular intervals using a held-out validation dataset.

For differentiable loss functions $\mathcal{L}$, such as the cross entropy loss, iterative gradient-based optimization methods can be employed. Therefore, the parameters are initialized as θ_0, typically with (small) randomly sampled values[5]. The first-order method of gradient descent then updates the parameters according to

$$\begin{aligned}\theta_{k+1} &= \theta_k - \alpha \nabla_\theta \left(\frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f(x_i, \theta_k), y_i) \right) \\ &= \theta_k - \frac{\alpha}{N} \sum_{i=1}^{N} \nabla_\theta \mathcal{L}(f(x_i, \theta_k), y_i),\end{aligned} \tag{2.8}$$

where α is the step length or *learning rate*. However, computing the gradient over the entire training dataset is computationally expensive and often leads to suboptimal generalization performance [29]. One can instead apply *stochastic gradient descent*, where a single example is selected at random for each iteration. In practice, this is achieved by randomly shuffling the dataset and then iterating through it, performing one parameter update per example according to

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta \mathcal{L}(f(x_i, \theta_k), y_i), \quad i \in \{1, ..., N\}. \tag{2.9}$$

Similarly to how the empirical risk is viewed as an estimate of the risk, the gradients computed using one or more examples can be viewed as estimates of the true risk gradient. As long as previously unseen examples are used, these estimates are unbiased. Thus, using a single example for each gradient estimation means that N updates can be performed without introducing any bias to the network. However, the estimates obtained using single examples have very high variance. It is therefore common to apply *mini-batch gradient descent*, where some number $m \in (1, N)$ of examples are used in each iteration. The number m is referred to as the *mini-batch size*, or simply batch size. There are many variables to consider when choosing the batch size, including the specific task and dataset as well as the implementation of the algorithm and the hardware it is run on. The reader is referred to [11] for details on this subject.

2.2.4.3 Backpropagation

In order to update the parameters using gradient-based methods, one must of course be able to compute the gradients

$$\nabla_\theta \mathcal{L}(f(x; \theta), y) = \frac{\partial}{\partial \theta} \mathcal{L}(f(x; \theta), y)).$$

[5]The randomness is necessary as it breaks the symmetry of the network. Without this asymmetry, groups of neurons may perform the same computations and thus also be updated equally.

Suppose for simplicity that the network f consists of a single sequence of layers $f_j(x_j;\theta_j)$, for $j = 1, ..., d$. The network function can then be broken down into a composition of the layer functions:

$$f = f_d \circ f_{d-1} \circ ... \circ f_1,$$

where $(g \circ h)(x) = g(h(x))$. Thus, expressions for the required gradients can be obtained by repeated application of the chain rule

$$(g \circ h)' = (g' \circ h) \cdot h'.$$

In practice this is done using the *backpropagation* algorithm. Given the network output $y_d = f_d(x_d, \theta_d)$ and the supervisory signal y, one can compute

$$\frac{\partial \mathcal{L}}{\partial y_d} = \frac{\partial}{\partial y_d}\mathcal{L}(y_d, y).$$

That is, the gradient of the loss with respect to the output y_d of the last layer. Using the chain rule, the gradient is then propagated to the input x_d and parameters θ_d of the layer according to

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial x_d} &= \frac{\partial \mathcal{L}}{\partial y_d}\frac{\partial y_d}{\partial x_d}\\ \frac{\partial \mathcal{L}}{\partial \theta_d} &= \frac{\partial \mathcal{L}}{\partial y_d}\frac{\partial y_d}{\partial \theta_d}.\end{aligned}$$

Now, the input x_d to the last layer is the output y_{d-1} of the previous layer. This procedure can thus be repeated, layer by layer, until the gradients are obtained for all parameters. All that is required to do this is the ability to differentiate the output of each layer with respect to its inputs and parameters.

As a detailed example, consider the convolutional layer described in (2.4). In the backward pass, the derivatives of the loss with respect to the layer outputs are given as tensor of shape $D \times W$ with elements $\nu_{d,i} = \partial\mathcal{L}/\partial y_{d,i}$. Using the chain rule, each of these derivatives can be propagated to any given input element or parameter. The derivative of the loss with respect to the input element or parameter is then obtained as a sum over all such contributions. Thus, the derivatives with respect to the kernel weights are given by

$$\frac{\partial \mathcal{L}}{\partial \omega_{d,c,j}} = \sum_{d'=1}^{D}\sum_{i=1}^{W}\frac{\partial y_{d',i}}{\partial \omega_{d,c,j}}\frac{\partial \mathcal{L}}{\partial y_{d',i}} = \sum_{i=1}^{W} x_{c,i+k-j}\nu_{d,i}. \tag{2.10}$$

In similar fashion, the derivatives with respect to the inputs take the form

$$\frac{\partial \mathcal{L}}{\partial x_{c,i}} = \sum_{d=1}^{D}\sum_{j=1}^{W}\frac{\partial y_{d,j}}{\partial x_{c,i}}\frac{\partial \mathcal{L}}{\partial y_{d,j}} = \sum_{d=1}^{D}\sum_{j=1}^{k}\omega_{d,c,j}\nu_{d,i+j-n-1}, \tag{2.11}$$

for $i = n+1, ..., n+W$ (excluding the zero-padding in the input). Lastly, the derivatives with respect to the bias parameters are given by

$$\frac{\partial \mathcal{L}}{\partial b_d} = \sum_{d'=1}^{D}\sum_{i=1}^{W}\frac{\partial y_{d',i}}{\partial b_{d'}}\frac{\partial \mathcal{L}}{\partial y_{d',i}} = \sum_{i=1}^{W}\nu_{d,i}. \tag{2.12}$$

2.2.4.4 Regularization

In this context, regularization refers to any strategy used to improve the generalization performance of a model, possibly at the expense of increased training loss. There are a large number of such strategies [11]. This section outlines a few popular choices.

The simplest regularization technique is known as *early stopping*. This refers to the act of discontinuing the training procedure when the validation loss starts to increase, even though the training loss may be steadily decreasing. That is, training is stopped before the model starts to overfit. In practice, this is usually done by storing the parameters that achieve the best performance on the validation dataset. After training, the model is returned to these parameters in the hope that the test performance is better as well.

Another type of regularization strategy is to limit the capacity of the model by penalizing large parameter values during training. For instance, *weight decay* or L^2 regularization adds a penalty term $\frac{\lambda}{2}\|\theta\|_2^2$ to the objective function. This effectively drives the parameters toward zero, with a strength controlled by the parameter $\lambda > 0$.

A popular form of regularization in deep neural networks is *dropout*. Here, the neurons in certain layers are “dropped out” with some probability at each training iteration. That is, their incoming and outgoing connections are ignored. In this way, random reduced networks are trained with each iteration. At test time, the full network is used, with the output of neurons rescaled according to the probability that they were present during training [30].

Lastly, we outline the method of *batch normalization*. This is not purely used as a regularization technique, although it has been shown to have a regularizing effect [31]. Batch normalization transforms the inputs to certain layers. During training, each input value is standardized, using the sample mean and standard deviation computed over the mini-batch, and subsequently scaled and translated by learnable parameters. At inference time, population statistics are used to standardize the inputs. These are often tracked by moving averages of the batch statistics computed during training.

2.3 The Adaptive Particle Representation

The Adaptive Particle Representation (APR) is a multi-resolution image representation, devised by Cheeseman et al [7] to relieve bottlenecks in the processing and storage of large microscopy images. It achieves this by replacing pixels with particles positioned according to the contents of the image. This section briefly motivates and introduces the APR.

2.3.1 Motivation

Modern fluorescence microscopy techniques allow scientists to image biological specimens at high resolution in both space and time [5]. This provides image data that are vital for the study of e.g. developmental processes in biology [32]. However, in order to acquire quantitative data from the output image datasets, various image processing techniques must be applied to, for instance, detect, count and track biological structures such as cells. This poses technical challenges, as the processing of potentially terabyte-sized datasets of 3D images through time often proves to be a severe bottleneck [5].

Cheeseman et al posit that the heart of the problem is not the amount of information contained in the images, but rather how that information is encoded as a regular grid of pixels. Due to the uniformity of the pixel representation, the sampling resolution in each dimension, spatial or temporal, is set everywhere according to the smallest inherent length scale in the process under study. This results in redundancies as, for instance, uniform background areas that convey little or no information are represented at this resolution as well. The APR reduces such redundancies by adaptively resampling the image based on local information content, as shown in Figure 2.8.

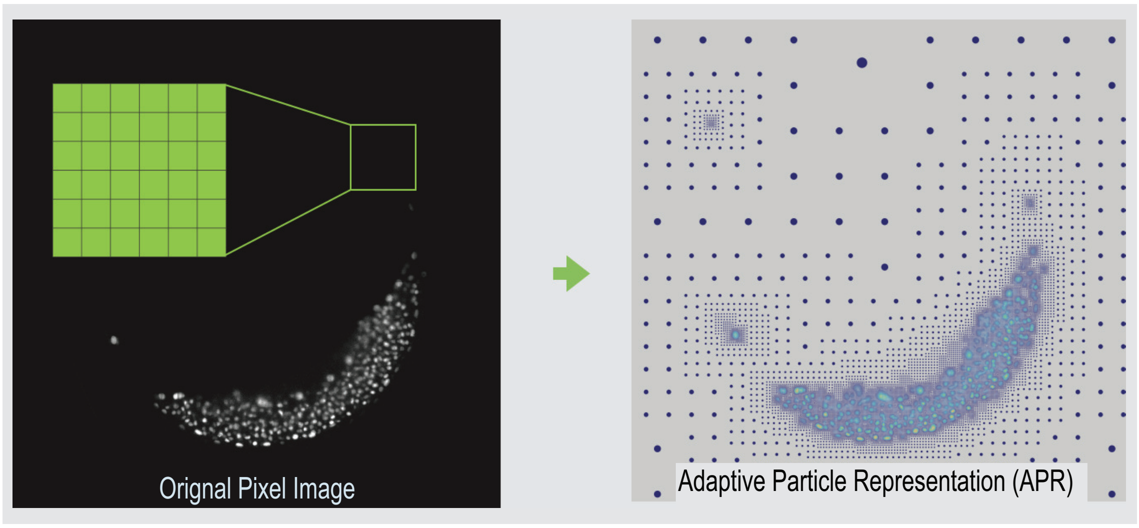


Figure 2.8: (left) Pixel image of cell nuclei in a zebrafish embryo (source: Gopi Shah, Huisken lab, MPI-CBG). The sparsity in the image gives rise to redundancies when represented by pixels. (right) Direct visualization of the APR of the image. Regions with little information content, or larger inherent length scale, are downsampled to reduce the redundancy of the representation.

2.3.2 How it works

The APR takes a regularly sampled input function I, such as a pixel image, and represents it using set of *particle cells* $\mathcal{V}$ and function values stored at particle collocation points $\mathcal{P}$. The particle cells partition space, and are defined by an integer *level* l specifying the size of the cell, and a multi-index i specifying its location. Consider a hyperrectangular image domain $\Omega \subset \mathbb{R}^d$ with maximum side length Ω_0. Then the particle cell $c_{i,l}$ occupies the spatial domain

$$s(c_{i,l}) = \prod_{k=1}^{d} [i_k \frac{\Omega_0}{2^l}, (i_k + 1)\frac{\Omega_0}{2^l}). \tag{2.13}$$

The set of particle cells can thus be written as $\mathcal{V} = \{\{c_{i_p,l_p}\}_{p=1}^{N_p} | i_p \in \mathbb{N}^d, l_p \in \mathbb{N}\}$. Note here that Ω_0 is assumed to be rounded up to the nearest power of two, and the levels are capped at $l_{max} = \log_2 \Omega_0$. In this way, particle cells at the maximum level l_{max} coincide with the pixels; particle cells at level $l_{max} - 1$ coincide with groups of 2^d pixels, and so on. Moreover, the particle collocation points x_p are taken to be the centers of the particle cells in $\mathcal{V}$, and the particle set $\mathcal{P} = \{\{I(x_p)\}_{p=1}^{N_p}\}$ holds intensities resampled at these points.

Now, the set of particle cells defines a piecewise constant *implied resolution function* $R^*(y)$, that takes value $2^{-l}\Omega_0$ for $y \in s(c_{i,l})$. The function value at any point $y \in \Omega$ can be reconstructed according to

$$\hat{I}(y) = \sum_{x_p \in \mathcal{N}(y, R^*(y))} \xi_p I(x_p), \tag{2.14}$$

where $\mathcal{N}(y, R^*(y)) = \{x \in \Omega : |x - y| \leq R^*(y)\}$ and the coefficients satisfy $\xi_p \geq 0$ and $\sum_{\mathcal{N}} \xi_p = 1$. That is, the reconstructed value is allowed to be an weighted average of particle intensities within $R^*(y)$ distance of y. Formation of the APR is formulated as the problem of finding the largest everywhere $R^*(y)$ (particle cells) such that the original image can be reconstructed within some user-set error bound. This is referred to as the *reconstruction condition*, stated formally as requiring that

$$\left\| \frac{I - \hat{I}}{\sigma} \right\|_\infty \leq E, \tag{2.15}$$

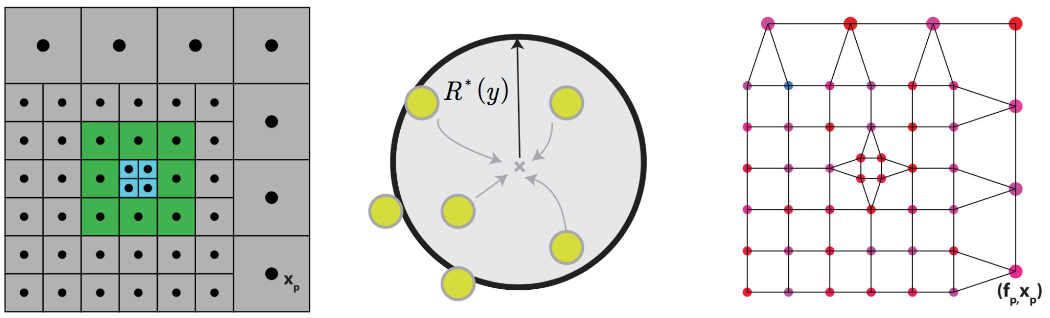
where the norm $\|h\|_\infty = \max_x |h(x)|$ is taken over all original pixel locations x. Here I are the original pixel intensities and $\hat{I}$ are the corresponding values reconstructed from the particles according to (2.14). The denominator, σ, is referred to as the *local intensity scale*. It is essentially a measure of the (local) contrast range in the image. By allowing σ to be spatially varying, effective adaptation in both bright and dim regions can be achieved [7]. Lastly, E is the user-set error threshold.

In practice, a sufficient condition for (2.15) is used, and the problem is reformulated in terms of the local intensity scale σ and the image gradient ∇I. Exactly how to compute these quantities is a design choice. The original implementation introduced in [7] uses smoothing B-splines to estimate the gradient, and a type of local standard deviation filter to compute the local intensity scale. These quantities are then input to a novel algorithm called the pulling scheme, which finds the optimal particle cell set $\mathcal{V}$. After this, all that remains is to interpolate the intensity values to the particle locations. Importantly, the entire pipeline from pixel image to APR has computational and memory costs proportional to the number of pixels in the image.

For additional details and a full derivation of the APR formation pipeline, the reader is referred to [7]. The following sections give a more intuitive view of the APR and how it can be interpreted and used in image processing tasks.

2.3.3 Interpretations of the APR for image processing

Just like pixel images, the APR can be interpreted in various ways to develop image processing methods. Taken at face value, the image is partitioned into regions (particle cells) of different sizes and each cell is assigned a particle carrying an intensity value. In this way, the particles can be viewed as a generalization of pixels; see Figure 2.9a. Moreover, using the reconstruction formula (2.14), intensity values can be reconstructed at any location to obtain a continuous function approximation, as illustrated in Figure 2.9b. If neighbor relations are defined, for instance between face-connected particle cells, the APR can be interpreted as a graph. The particles then constitute the vertices of the graph and edges are drawn between neighboring particles. This is shown in Figure 2.9c.



(a) Particle cells and collocation points **(b)** Continuous reconstruction **(c)** Neighbor graph

Figure 2.9: Examples of interpretations of the APR for image processing (reproduced from [7]). (a) illustrates the particle cells (squares) and particles (dots). (b) shows how intensity values can be obtained at arbitrary points as weighted averages of particles (yellow dots) within a radius defined by the implied resolution function. (c) The APR interpreted as a graph, where the particles are the vertices and edges are drawn between particles belonging to face-connected particle cells.

All of these interpretations can be used to define image processing techniques using the APR. In this work, a fourth interpretation of the APR as a tree structure is used extensively. It is described in detail below.

2.3.3.1 Tree interpretation

The tree interpretation of the APR is best explained with a simple example. Suppose we have a one-dimensional image comprised of eight pixels, and consider the set of all possible particle cells. At the maximum resolution level $l_{max} = 3$, there are eight possible particle cells that coincide with the pixels. One resolution level lower, $l = 2$, has four possible particle cells, each spanning two pixels. Similarly, at $l = 1$ there are two possibilities and at $l = 0$ the entire image domain is represented by a single particle cell. The spatial relationships between particle cells at different levels give rise to a binary tree structure, as shown in Figure 2.10. This extends easily to higher dimensions, with quadtrees in 2D or octrees in 3D.

In the pixel representation of the image, the maximum resolution $l_{max} = 3$ is chosen everywhere. Thus, the pixels constitute the leaf nodes of the full binary tree. The APR, on the other hand, selectively downsamples certain regions of the image, representing them at lower resolution levels. In this way, the APR constitutes the

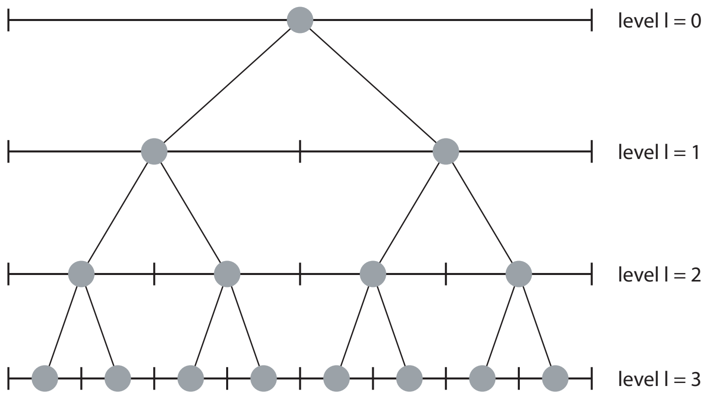


Figure 2.10: Possible particle cells (intervals) and particles (gray dots) for a 1D image with 8 pixels. Links are drawn between particle cells at different levels that occupy the same space, forming a binary tree structure.

leaf nodes of a pruned version of the tree, as shown in Figure 2.11. There is a clear correspondence between the pixel representation and APR when viewed in this way. The main difference is that the pixel representation is restricted to a single resolution level, while the APR is not. From this perspective, the pixel representation can be viewed as a special case of the APR.

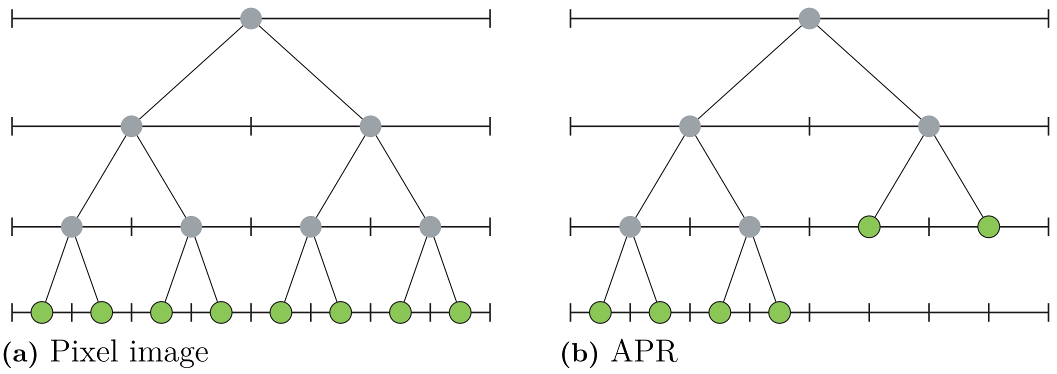


Figure 2.11: Binary tree interpretation of a 1D pixel image (a) and a corresponding APR (b). Formation of the APR can be viewed as a pruning of the full binary tree, where certain regions are downsampled and represented by ancestral nodes at lower resolution.

The tree interpretation relates the APR to several existing concepts in image processing, including wavelet decompositions [33] and image pyramids [34]. An image pyramid can be obtained from a pixel image by applying successive downsampling operations. When downsampling by factors of two, the image pyramid coincides with the tree structure depicted in Figure 2.10. That is, each level of the tree (or pyramid) represents the image at a certain resolution. Essentially, the APR is formed by selecting a partition of spatially non-overlapping elements from the pyramid. Hence, each resolution level of the APR constitutes a sparse (incomplete) version of the image at the corresponding level of the pyramid.

2.3.4 Data structures

The APR and related functionality is implemented in C++ and freely available through the open-source software library LibAPR [35]. Full details on the APR and its implementation can be found there. Here, a brief description of the data structures utilized in this work is given.

Recall that the APR consists of two sets: the particle cell set $\mathcal{V}$, which defines the spatial locations and resolution levels of the particles, and the particle properties $\mathcal{P}$. These are kept separate in the *sparse APR* data structure used to store and access the APR data. The particle properties are stored as a contiguous array, whereas the spatial location and resolution information is encoded as a separate "access" class. This is done by separating the particle cells by level, as illustrated in Figure 2.12, and encoding each level as a sparse matrix in a format similar to *compressed row storage* [36]. More precisely, for each non-empty level l and spatial index x, each contiguous block of occupied particle cells in the y-direction is encoded by its first and last y-index, as well as a global index for the first element in the block. These global indices are used to access the particle properties. In this way, particles can be accessed by specifying the level and x-index, and iterating over the sparse y dimension using a special iterator.

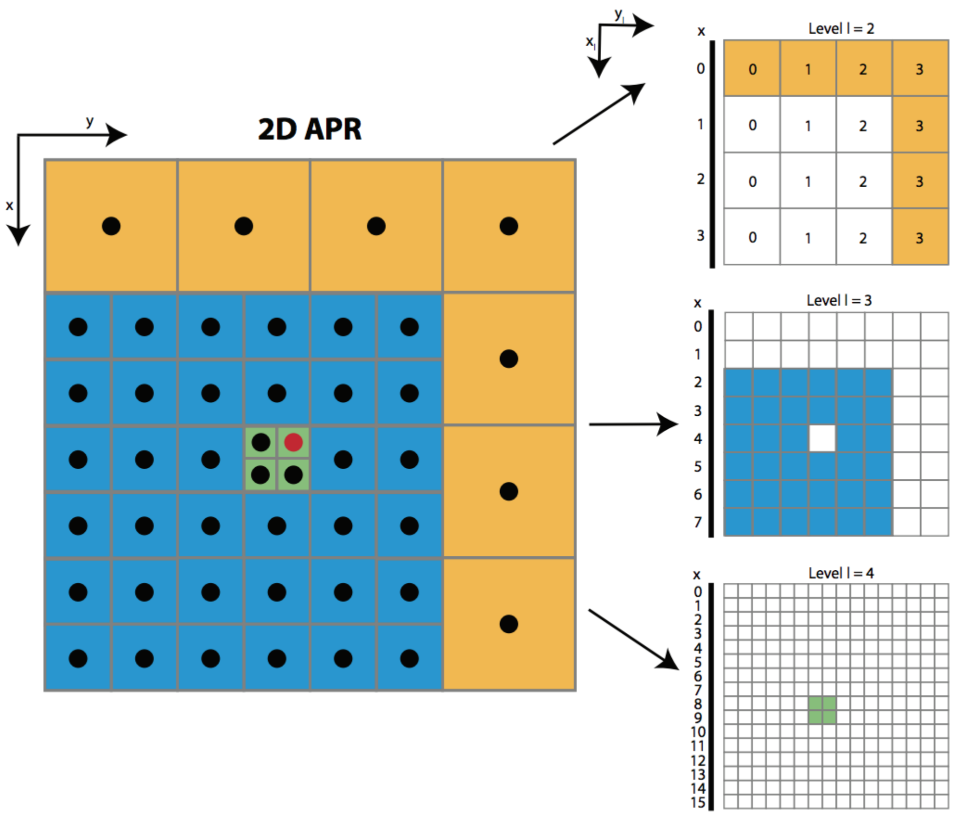


Figure 2.12: The sparse APR data structure separates the data, first by level and then by spatial index in the x-direction.

In order to use the tree interpretation for processing tasks, particle cells that are not part of the APR must be accessed. This can be done by "filling" and storing the interior of the tree, i.e. the ancestors of the APR nodes, as a separate instance of the sparse APR data structure. We call this set of nodes the APRTree. Figure 2.13 illustrates the distinction between the APR and APRTree.

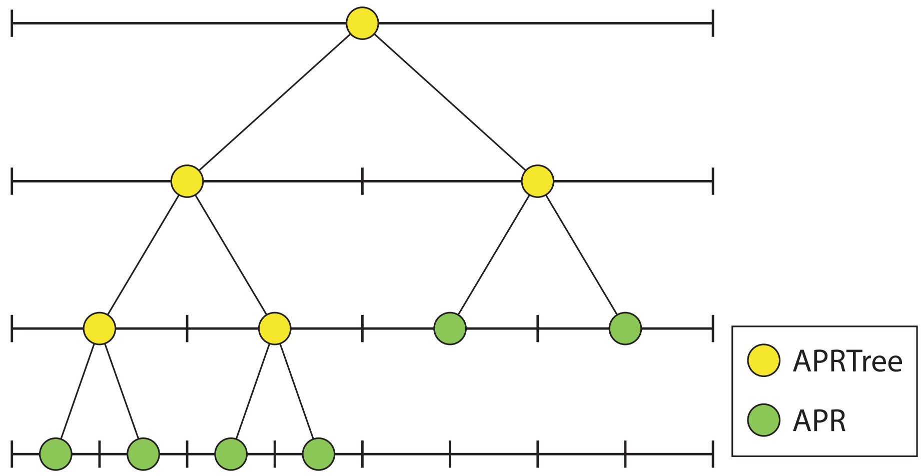


Figure 2.13: The APR constitutes the leaf nodes of the tree, whereas the APRTree comprises all ancestors of the APR nodes. Both of them can be stored and accessed using separate instances of the sparse APR data structure.

3
APRNet

The main objective of this work is to design an extension of convolutional neural networks to allow direct operation on the APR. This would first of all allow users of the APR to apply deep learning methods without the need to revert to pixel images. Furthermore, it may be potentially useful for deep learning practitioners by reducing the computational costs of processing large and sparse images.

In order to extend any given neural network architecture to the APR, its layer operations must be modified to allow operation on the APR data structure. Here, focus is placed on the two most commonly used operations in CNNs for image classification, namely convolution and pooling. It is worth mentioning that the various interpretations of the APR give rise to several alternatives for extending these operations. The extension of CNNs to graphs, point clouds and irregular or non-Euclidean data in general are popular topics of current research [37]. Published work in this area may provide direct approaches for certain interpretations of the APR. However, the APR maintains a close connection to the original pixel image through the reconstruction condition (2.15). Since CNNs are best-established for pixel images, it is reasonable to attempt to leverage this fact. Therefore, the tree interpretation is used. As discussed in Section 2.3.3.1, this allows the APR to be viewed simply as a multi-resolution image, or as a sparsely populated image pyramid.

This chapter first features a discussion of related work on the use of convolutional neural networks for multi-scale image pyramids and sparse image data. The methods developed for the APR are then described in detail, followed by notes on implementation and a brief discussion of computational and memory costs.

3.1 Related work

Image pyramids are useful in a wide range of image processing tasks [34], and several works make use of them in deep learning methods. LeCun et al [38] perform semantic segmentation by applying a single CNN to multiple levels of a Laplacian pyramid constructed from the input image. The resulting feature maps at different scale are then concatenated, with the coarser maps upsampled to match the finest-scale map, and used to predict the pixel labels. Similar approaches are used for classification in [39] and [40]. Yoo et al [39] propose a *Multi-scale Pyramid Pooling* (MPP) layer to combine the different-scaled feature maps for general recognition. Nam et al [40] introduce the *pyramid-based scale-invariant* CNN (PSI-CNN) architecture for facial recognition. Several pyramid levels are convolved independently and pooling is performed on the highest resolution features. The pooled features are then concatenated to the features at the lower level, and the resulting stack is combined using convolution operations.

Note that the image pyramid representation of an image with N elements in d dimensions, when downsampling by factors of 2, contains up to $2^d N/(2^d - 1)$ elements. The computational and memory requirements for processing such a pyramid are thus increased accordingly. Other works attempt to make use of multi-scale representations while limiting the additional computations. For instance by applying a CNN to the original image and extracting pyramids of feature maps from different stages of the network [41, 19], which are then used to form the final predictions.

Sparsity in data can be leveraged to increase the efficiency of processing tasks. Riegler et al [42] introduce the OctNet architecture for sparse 3D data. The domain is divided into regions of 8^3 voxels, which are further subdivided into shallow octrees depending on their information contents. In this way, the data is represented as a grid of shallow octrees. Low-resolution nodes in the octrees are viewed as groups of voxels that share the same value. Regular voxel convolutions are performed, but reimplemented to exploit the lower resolution tree nodes in order to reduce the amount of floating point operations. Similarly, pooling is performed on the highest resolution nodes and groups of octrees are merged to maintain the maximal depth.

Graham [43] implemets efficient algorithms to apply convolutional neural networks to sparse image data by reducing the amount of redundant computations. Essentially, each layer has a “ground state”, corresponding to its output when the network receives all-zero input. Then, as non-trivial inputs are processed, hidden neuron values only need to be computed where they differ from the ground state. The computational workload is thus decreased for sparse inputs.

To summarize, previous works apply convolutional neural networks to multi-scale pyramid representations of images by processing each pyramid level independently. Feature maps or results at different scales may then be downsampled (pooled) or upsampled to matching scales and combined to form the final prediction. Sparsity in data is leveraged to reduce the computational burden by identifying and filtering out redundant computation.

3.2 Layer operations for the APR

In order to extend convolutional neural networks to the APR, the layer operations, here convolution and pooling, must be defined as mappings from APR to APR. Importantly, to allow the APR to have a beneficial impact on computational and memory costs, these should scale with the number of particles and not depend on the original image size. The following sections detail one way of achieving this, and how the resulting methods relate to the works described above.

3.2.1 Pooling

Using the tree interpretation of the APR, pooling can be done conveniently by aggregating values of sibling nodes and inserting the result into the parent node. At the maximum resolution level, this corresponds to pooling of pixels in windows of size 2^d, where d is the number of spatial dimensions, with a stride of 2. Since the

APR forms a spatial partition of the image domain, the maximum level particles always exist in such groups. Moreover, since lower resolution particles are already downsampled, it is reasonable to only pool the highest-resolution particles. Intuitively, this can be viewed as a simple tree operation, where the maximum level nodes are "pulled" up to their ancestral nodes, as depicted in Figure 3.1.

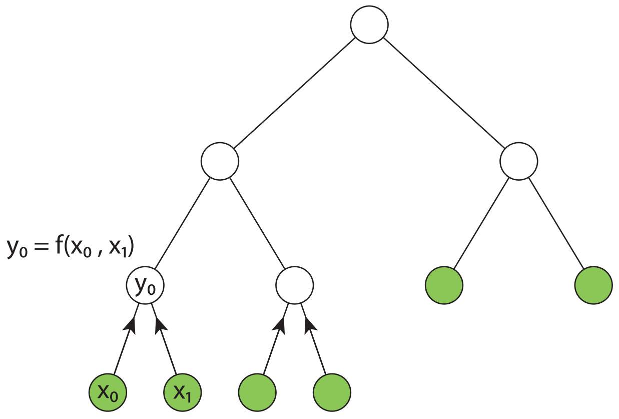


Figure 3.1: One-dimensional example of pooling on the APR as a tree operation. Sibling nodes (particles) at the maximum resolution level are aggregated by a generic function f, and the results are inserted into the parent nodes.

This is similar to how pooling is performed on image pyramids in the PSI-CNN [40]. That is, the highest resolution feature maps are pooled and joined with the features at a lower resolution. The main difference is that there is no spatial overlap between the pooled APR particles and those already occupying the target level. Hence, the feature maps can be joined directly rather than concatenated and integrated by convolutions. Although such an integration step could be added to any network design for the APR as well, if deemed useful.

Consider now the APR pooling operation, as illustrated in Figure 3.1, with f being the maximum operation. The first output feature, y_0, is then computed according to

$$y_0 = \max(x_0, x_1).$$

In the backward pass, the corresponding pooling layer receives $\partial L/\partial y_0$ as input, which must be propagated to the inputs x_0 and x_1. Utilizing the chain rule, this is easily done through

$$\frac{\partial L}{\partial x_0} = \frac{\partial L}{\partial y_0}\frac{\partial y_0}{\partial x_0} = \frac{\partial L}{\partial y_0}1_{\{x_0 \geq x_1\}} \tag{3.1}$$

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial y_0}\frac{\partial y_0}{\partial x_1} = \frac{\partial L}{\partial y_0}1_{\{x_0 < x_1\}} \tag{3.2}$$

where $1_{\{A\}}$ is an indicator variable for the statement A. That is, it takes value 1 if A is true and 0 otherwise. Note that the case $x_0 = x_1$ is somewhat ambiguous as, strictly speaking, the derivatives of the max function do not exist at such points (the left and right derivatives exist but are not equal). With subderivatives in mind, propagating the error to x_1, x_2 or dividing it between the two are all valid options. However, the choice is of little practical importance as the situation is extremely unlikely when working with floating-point numbers.

3.2.2 Convolution

There are several options available for extending the convolution operation to the APR. For instance, the variant used in OctNet [42] could be applied with minimal modification. This would be equivalent to performing a regular convolution operation on the fully reconstructed pixel image and average pooling the result back to the particles. While such an approach is likely to yield results comparable to those of traditional (single-scale) CNNs, the computational complexity depends on the original image size. Moreover, the different resolution scales of the APR carry potentially useful information that would not be utilized.

To allow the network to take advantage of this information, the convolution operation should be performed at different scales. In line with previous works on CNNs for image pyramids, each resolution level of the APR could be processed independently using (sparse) convolutions. However, since the resolution levels of the APR only constitute incomplete images, processing them in a completely independent manner would again result in suboptimal use of the available information. The convolution operation should therefore be able to make use of neighborhood information also at different resolution levels.

Therein lies a problem, as the neighborhoods are anisotropic with variable structure that depends on the adaptation. However, using the formula (2.14), values can be reconstructed at any location. Hence, rather than defining a convolution operation that allows for anisotropic interactions, an intermediate reconstruction step can be used to temporarily isotropize the neighborhoods. This is shown in Figure 3.2. Regular discrete convolutions can then be performed on the reconstructed isotropic patches.

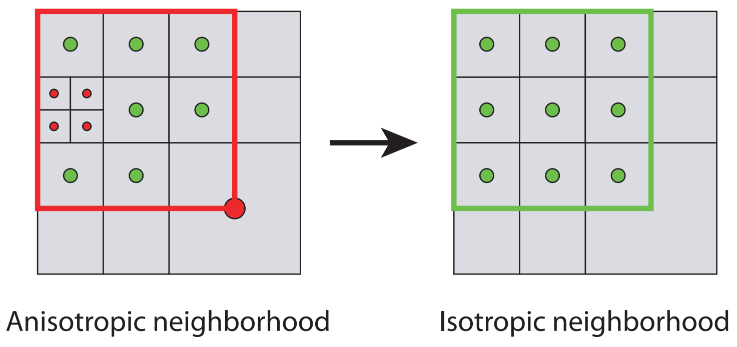
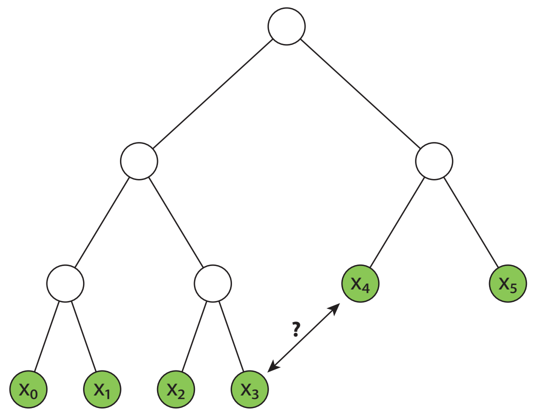
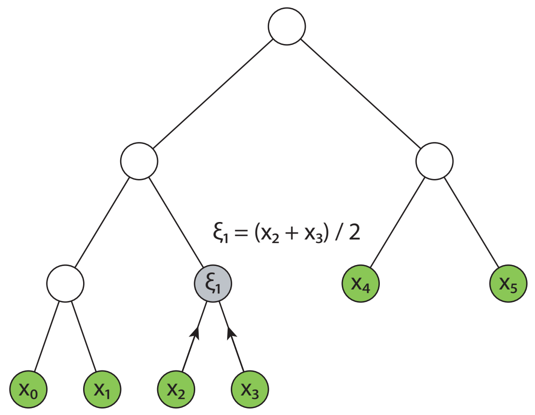


Figure 3.2: Using the reconstruction formula (2.14), information can be interpolated between resolution levels to form isotropic neighborhoods.

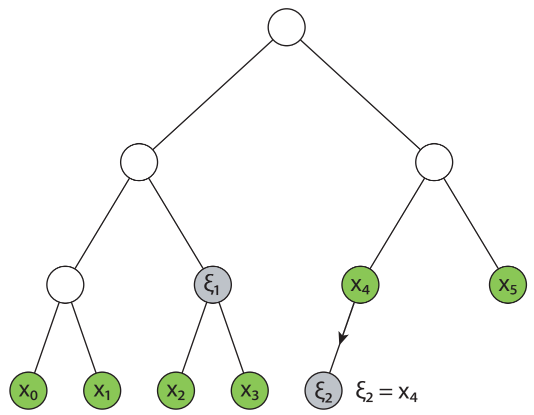
There is some freedom in choosing exactly how to interpolate information between resolution levels. High-resolution particles can be downsampled to obtain values at lower resolution levels, similar to the pooling operation described in the previous section. Due to the symmetry of the problem it is reasonable to simply take the average value of the descendant nodes. To interpolate from low to high resolution, low-level particle cells can be viewed as regions of constant value spanning several higher-level cells. Descendant nodes in the tree can therefore directly inherit the value of the parent. This is referred to as *piecewise constant upsampling.* Importantly, both average downsampling and piecewise constant upsampling are always valid reconstructions with respect to the reconstruction condition (2.15). Figure 3.3 illustrates the interpolation steps as tree operations using a simple 1D example.



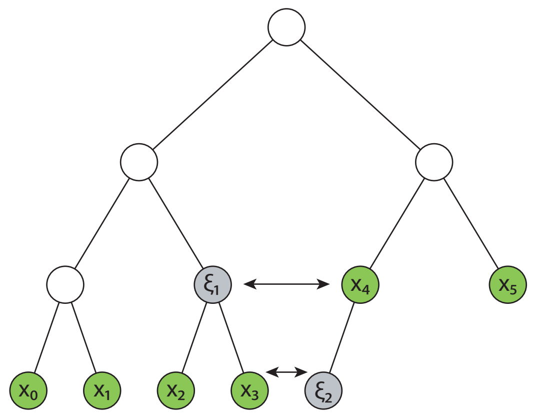
(a) Anisotropic neighborhoods



(b) Downsample high-resolution information



(c) Upsample low-resolution information



(d) Locally isotropic neighborhoods

Figure 3.3: Illustration of the steps used to interpolate information between resolution levels. In this way, isotropic neighborhoods are temporarily reconstructed around each particle, allowing for isotropic interactions.

This technique allows the full image pyramid to be reconstructed from the APR. Hence, regular discrete convolutions can be applied at any resolution and any location, using arbitrary kernel sizes. However, for practical reasons, the structure of the APR should be preserved by the convolution operation. That is, output values should only be computed at the actual particle locations. This can be achieved by only applying the convolutions where the kernel center coincides with a particle, with zero padding and reconstructions applied as necessary to complete the neighborhoods. Effectively, this corresponds to applying a type of sparse convolution to each level of the sparse image pyramid defined by the APR. Note also that the kernel parameters can be allowed to differ between the resolution levels. Figure 3.4 illustrates the procedure, using the isotropic neighborhoods constructed in Figure 3.3.

In order to formalize the above procedure and build it into a neural network

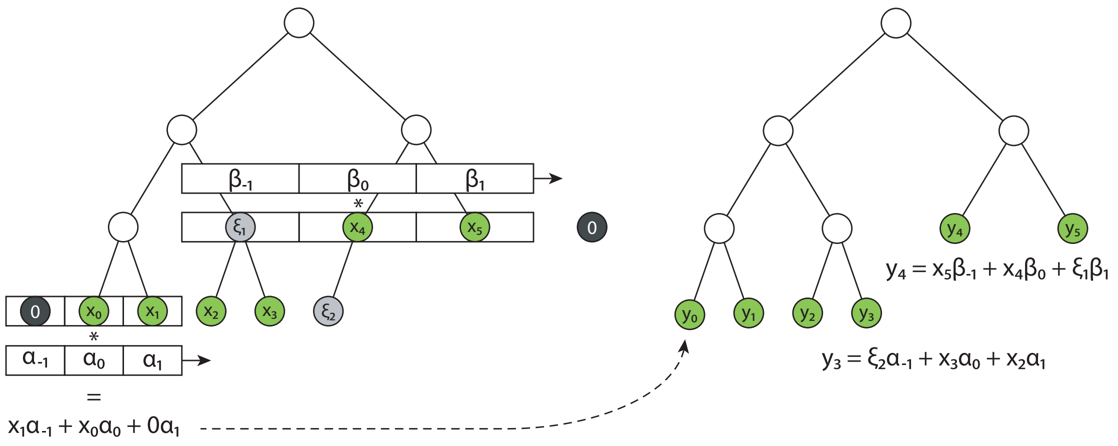


Figure 3.4: Using the reconstructed isotropic neighborhoods and zero padding, convolutions can be applied independently to each resolution level, with the kernels centered on each APR particle.

layer, consider again a one-dimensional example. Let $\hat{I}_{c,i,\ell}$ denote the value of cth channel of the particle with spatial index i at level ℓ. If X_ℓ denotes the set of spatial indices where particles exist at level ℓ, the value $\hat{I}_{c,i,\ell}$ corresponds to a particle value if $i \in X_\ell$, and a reconstructed value otherwise (or 0 if the location is outside of the image domain). Suppose that there are C input channels, and define D convolution kernels of (odd) spatial size K. Allowing the kernel parameters to vary between resolution levels, the kernel bank is given by a 4D tensor with elements $\omega_{d,c,k,\lambda(\ell)}$, where λ maps levels to tensor indices. The output of the layer is a tensor with elements of the form

$$O_{d,i,\ell} = \sum_{c,k} \hat{I}_{c,i-k,\ell}\,\omega_{d,c,k,\lambda(\ell)}, \tag{3.3}$$

computed for output channels $d = 1, ..., D$ and all spatial indices $i \in X_\ell$ for each level ℓ. The sum is taken across all input channels and the spatial extent of the kernel, with indexing such that the center element coincides with $\hat{I}_{c,i,\ell}$.

In the backward pass, the gradient of the loss $\mathcal{L}$ with respect to the layer output is given as a tensor with elements

$$dO_{d,i,\ell} := \frac{\partial \mathcal{L}}{\partial O_{d,i,\ell}}.$$

Using the chain rule, these derivatives can be propagated to the kernel weights according to

$$\frac{\partial \mathcal{L}}{\partial \omega_{d,c,k,l}} = \sum_{d',i,\ell} dO_{d',i,\ell} \frac{\partial O_{d',i,\ell}}{\partial \omega_{d,c,k,l}} = \sum_{\substack{i \in X_\ell \\ \ell \text{ s.t. } \lambda(\ell) = l}} dO_{d,i,\ell}\hat{I}_{c,i-k,\ell}. \tag{3.4}$$

Here, the sum is taken over all spatial indices of particles at each level where the corresponding kernel was used. In similar fashion, the derivatives can be propagated to the intensities $\hat{I}$ through

$$\frac{\partial L}{\partial \hat{I}_{c,i,\ell}} = \sum_{d,j,\ell'} dO_{d,j,\ell'} \frac{\partial O_{d,j,\ell'}}{\partial \hat{I}_{c,i,\ell}} = \sum_{d,k} dO_{d,i+k,\ell}\,\omega_{d,c,k,\lambda(\ell)}. \tag{3.5}$$

However, the $\hat{I}$-values include reconstructed particles not part of the input APR. To obtain the correct derivatives with respect to all of the originally input particles, some of the derivatives in (3.5) must be backpropagated further through the reconstruction operation. This is easily done, as the reconstructed values take the form of averages of one or more particle values. Hence, viewing the backward reconstruction as a tree operation, the derivative value at a temporary node is either added to an ascendant, or divided evenly among its descendants.

If bias parameters are used, they contribute to each output element by a constant amount. Therefore, the corresponding derivatives are easily computed as sums of the dO-values at the output positions they contributed to. Different parameter sharing schemes can again be used for the bias terms. For example, one might use one bias parameter per output channel, or one parameter per output channel and level.

3.3 Implementation

In order to utilize existing implementations of data structures and iterators, the operations have been implemented in C++ as part of the library LibAPR. Computations are performed by the central processing unit (CPU), and accelerated by shared-memory parallelism using OpenMP [44].

When it comes to building and applying neural networks, there are many deep learning frameworks and libraries that provide user friendly high-level APIs that simplify the process tremendously. However, most of these frameworks are Python-based. For this reason, the operations implemented in C++ have been wrapped to Python using pybind11 [45] and built into custom, high-level "layer" modules in the popular deep learning framework PyTorch [46]. This allows seamless design of APR-Nets, as well as the use of existing implementations of loss functions, optimization schemes and regularization techniques without modification.

This section briefly describes the communication between C++ and Python, and outlines the algorithmic implementation of the convolution and pooling operations.

3.3.1 Communication between C++ and Python

In the current implementation, three C++ classes are wrapped to Python. Two of the classes contain methods for basic APR usage, such as the setting of parameters and creation of APRs from Python arrays, as well as the retrieval of particle properties as Python arrays. All of the developed operations are wrapped in a separate class. The operations are called from Python as methods of an instance of this class. These methods take as input a number of Python arrays, containing APRs (as Python objects) as well as the operation inputs (particle properties), parameters and initialized output arrays. On the C++ side, the operations are implemented using pybind11 to accept Python arrays passed by reference. This allows direct manipulation of the data in C++, without any copy operations.

3.3.2 Max pooling algorithm

The forward max pooling operation, as described in Section 3.2.1, essentially boils down to finding the particles in the input and output that are connected in the tree. This is done using two iterators, one over the maximum level APR[1] particles and one over the APRTree particles at the level below. Lower level particles, which remain untouched by the operation, are simply copied to the output.

In the backward pass, the gradient with respect to the output is propagated to the input particles that constituted the maxima in the forward pass. This can be done by repeating the forward operation and registering the indices of the maxima. However, it is more efficient to do this in the original forward pass and storing the indices for the backward operation. In this way, the backward pass reduces to reading an index map and inserting values at the corresponding positions. Algorithms 1 and 2 outline the forward and backward operations utilizing this technique. Of course, when employing trained networks the forward pass can be made more efficient by not storing the indices.

```
Data: input
Result: output, index_map
compute size of output;
initialize output ($-\infty$) and index_map;
for level < max_level do
    for p in input at level do
        copy p to output;
        insert index of p into index_map;
    end
end
for p in input at max_level do
    find parent node of p;
    if p > output at parent position then
        update output at parent position to p;
        update index_map at parent position to index of p;
    end
end
```

Algorithm 1: Outline of the forward pass of the max pooling operation on the APR. The index map is kept to increase the efficiency of the backward pass during training.

It is clear that the computational cost of the backward pass is linear in the number of output particles. In the forward pass, the input particles are iterated over exactly once. One comparison is performed for each maximum level particle, whereas particles at lower levels only require an insertion. The exact number comparisons or insertions depend on the precise adaptation, but they are clearly bounded by the number of particles. Therefore, the computational cost of the forward pooling operation is linear in the number of input particles.

[1]and possibly also tree particles, if pooling operations have been applied previously.

```
Data: $grad\_out$, $index\_map$
Result: $grad\_in$
initialize $grad\_in$;
for $i <$ size of $index\_map$ do
    insert $grad\_out$ at $i$ into $grad\_in$ at $index\_map[i]$;
end
```

Algorithm 2: The backward pass of the max pooling operation on the APR. This operation is trivial when storing the index map in the forward pass (see Algorithm 1).

3.3.3 Convolution algorithm

The convolution operation requires more care to be implemented in an efficient manner. It could be implemented as a simple loop over the particles, where the isotropic patch is computed at each location and convolved with the kernel. However, that would result in poor memory access patterns and cache efficiency. Instead, the current implementation performs the operation in blocks, sacrificing some temporary memory usage to improve the computational efficiency. Algorithm 3 outlines the procedure. First, the APRTree is filled by successively averaging the input particles. This allows for isotropic patches to be reconstructed rapidly, as the values are precomputed. The convolution operation is performed in a loop over the levels and spatial indices. For each level, a temporary image buffer is initialized. This buffer spans the entire sparse dimension (y), and its extent in the x-direction can be chosen anywhere from the size of the kernel to the entire image domain. In this way, isotropic patches can be reconstructed in bulk and stored in the temporary buffer for efficient access. The operation is performed by iterating over the x dimension, possibly updating one line of the buffer at each position, and the existing y indices for the given x and level. Convolution outputs are computed by applying the kernel to the corresponding locations in the temporary buffer.

```
Data: $input$, $kernel$, $bias$
Result: $output$
initialize $output$;
initialize and fill $APRTree$;
for $level < max\_level$ do
    initialize $temp\_img$;
    for spatial index $x$ at $level$ do
        update $temp\_img$ (insert values from $input$ and $APRTree$);
        for existing $y$ at $level$ and $x$ do
            convolve $kernel$ with $temp\_img$ at $(x, y)$;
            insert $result + bias$ into $output$;
        end
    end
end
```

Algorithm 3: Outline of the forward convolution operation on the APR.

To propagate gradients backward through the convolution operation, the forward pass is repeated with a few extra steps. Essentially, each part of the forward pass is done in reverse. The convolution outputs are computed by convolving the temporary image buffer. Hence, the gradients with respect to the kernel weights receive contributions of temporary buffer values at each particle location. The gradients with respect to the input particles are obtained in three stages. First, the gradients with respect to the temporary image values are computed and stored in an additional temporary buffer. This buffer is in turn updated in reverse by propagating its values to the input gradient if they correspond to existing particles, and to an additional APRTree structure otherwise. Finally, the reverse of the tree filling operation is used to propagate these values to the input APR particles. Algorithm 4 summarizes the procedure.

```
Data: input, kernel, grad_out
Result: grad_in, grad_kernel, grad_bias
initialize output gradient arrays;
initialize and fill APRTree;
initialize grad_APRTree;
for level < max_level do
    initialize temp_img and grad_temp_img;
    for spatial index $x$ at level do
        update temp_img;
        for existing $y$ at level and $x$ do
            dO = grad_out at current position;
            add dO times patch of temp_img at $(x, y)$ to grad_kernel;
            add dO times kernel to grad_temp_img;
            add dO to grad_bias;
        end
        propagate (parts of) grad_temp_img to grad_in and
          grad_APRTree;
    end
end
propagate grad_APRTree to grad_in;
```

Algorithm 4: Outline of the backward convolution operation on the APR.

It is difficult to exactly assess the computational complexity of the algorithms implemented in this way. Most of the floating-point operations take place in the innermost loops, i.e. to compute the convolution outputs or the corresponding gradients. Since the algorithm iterates over the particles exactly once, this requires $N_p(k^2 + 1)$ multiply-add operations in the forward pass and $N_p(2k^2 + 1)$ such operations in the backward pass, where N_p is the number of particles and k the kernel size. However, additional floating-point operations are required to fill and unload the APRTree. More precisely, four operations are required for each node in the tree. The exact number of tree nodes depend on the precise spatial structure of the APR, but is clearly no larger than the number of APR particles. Thus, it can be concluded that the computational costs of both the forward and backward pass are linear in the number of particles.

3.3.4 Notes

Implemented as neural network layers, batches of multi-channel inputs must be processed. This is achieved by additionally iterating over the batch, input channels and output channels. For all operations the outermost loop, over the APRs in the batch, is parallelized using OpenMP. Furthermore, three versions of the convolution operations are implemented. One version accepts kernels of arbitrary size, while the others are specialized for kernels of size 3×3 and 1×1, respectively. The specialized implementations employ explicit loop unrolling to speed up the innermost loops, and the 1×1 version additionally ignores all reconstruction steps.

To ensure that the gradients are computed correctly, the outputs of the backward operations have been compared to approximate values obtained by central finite differences. These values are exact for the convolution operation, since it is linear.

4 Results

In this chapter, we empirically evaluate and study properties of the APRNets in relation to traditional pixel CNNs. This is done by training and testing several architectures of both APRNets and pixel CNNs on benchmark image classification tasks. Focus is placed on the classification performance and learning characteristics of the networks, which is investigated using both synthetic and natural image datasets. Also of interest is the impact of different APR parameter settings on the performance of APRNets. This is studied by repeating certain experiments with different values of the error threshold E in the reconstruction condition (2.15). Lastly, the potential benefits of APRNets to computational and memory costs are discussed and empirically evaluated.

4.1 Synthetic image benchmarks

The use of synthetic images offers a high degree of control and freedom in the design of experiments. This section describes the implementation of a simple image generation pipeline, which is used to develop two experiments. First, the capacities of the networks are studied by training them using streams of data sampled directly from the image generating distribution. This effectively reduces the learning problem to a traditional optimization problem, and should expose fundamental shortcomings of the networks. It also provides an ideal setting in which the effects of the relative error threshold can be studied. In the second experiment, networks are trained and tested on fixed datasets, emulating real-world learning problems.

4.1.1 Image generation

Noise-free images containing one of six possible object types are generated. The object types are the contours of regular and "fuzzy" versions of three basic geometric shapes: circles, squares and triangles. Examples of the classes are shown in Figure 4.1. To generate the objects, the contours are parameterized in a continuous manner. Fuzzy objects are obtained by adding sinusoidal perturbations with random amplitude and frequency to the contour. The parameterized curves are then crudely mapped to pixel locations by rounding. This results in pronounced aliasing effects, which essentially constitute the difficulty of the task, as smaller perturbations can be masked behind them and challenging to detect. One object of random size, location and orientation is generated for each image. The intensity values of the background and foreground regions are uniformly sampled in $[0, 1]$, with a minimum difference of 0.05.

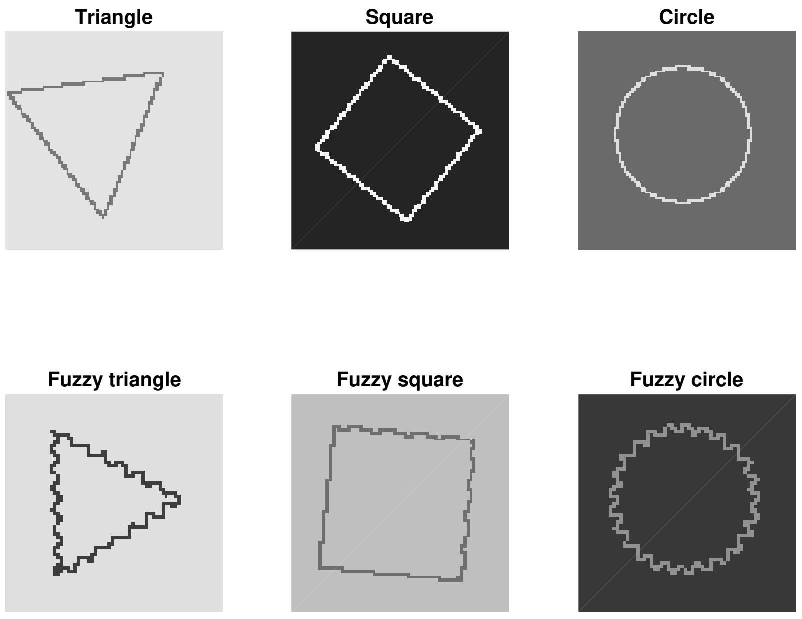


Figure 4.1: Examples of the regular and fuzzy shapes. Here, the random perturbations in the fuzzy triangle and circle have large amplitude and are clearly visible. The fuzzy square is an example of a more difficult case, as the perturbations are more subtle and partially lost in the rounding to pixels.

In all experiments described in subsequent sections, images of size 256×256 pixels are generated, with the maximum object size covering roughly one fourth of the image. The resulting images are extremely sparse, and in some sense optimal for the APR. Essentially, all the information is located in a band of pixels surrounding the object boundary. The remaining areas are constant and convey no useful information. Therefore, the APR can achieve perfect reconstruction using few particles by fully resolving the object boundary and aggressively downsampling the remaining areas of the image.

4.1.2 APR performance

Prior to building and training any models, the performance of the APR on the images is investigated, as well as the effects of different parameter settings. For the sake of simplicity, only one parameter is considered here, namely the relative error threshold E that bounds the reconstruction error. The remaining APR parameters are fixed at default values.

Two metrics are used to assess the performance of the APR. Its benefit is measured by the *computational ratio* (CR), defined as

$$\mathrm{CR} = \frac{\text{number of pixels}}{\text{number of particles}}.$$

Information loss due to the APR is assessed by looking at the reconstruction error. That is, the difference between the original pixel image and that reconstructed from the APR. In addition to these two metrics, images of other particle properties can be reconstructed. For instance, reconstructing an image where each pixel encodes the level of the particle cell it belongs to gives a more holistic view of the adaptation. We refer to such images as *level maps*.

The performance of the APR on the synthetic images is evaluated using a set of 1000 generated images. For each image, the APR is computed for different values of the error threshold E and the performance metrics registered. Figure 4.2 (A-B) summarize the results. It is seen that values of E around 0.2 or lower produce lossless APRs in most cases. The computational ratio varies significantly with e.g. object size, but is on average slightly above 50 for these settings. As E is increased, pixel level information around the object contours is gradually lost until eventually only downsampled information remains, which occurs around $E = 0.5$. This pattern then repeats itself as E continues to increase. An example of this effect is shown in Figure 4.2 (C-E). With $E = 0.1$, the APR captures all the information in the original image by retaining a band of pixel-resolution particles around the object contour, whereas for $E = 0.4$ only some of these particles are retained.

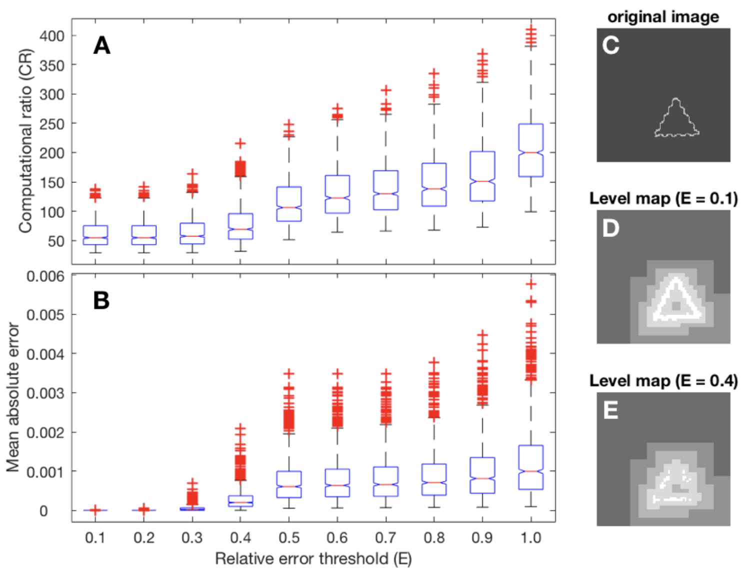


Figure 4.2: Summary of APR performance on synthetic images. (A-B): Box plots of computational ratios (A) and average absolute reconstruction errors (B) for 10 different values of the relative error threshold E, on a set of 1000 generated images. (C-E): Example image of a perturbed triangle (C), as well as level maps showing the adaptation of the APR for $E = 0.1$ (D) and $E = 0.4$ (E). Brighter pixels in the level maps indicate higher level particles, with white corresponding to pixel resolution.

4.1.3 Experiments

Two learning tasks are devised using the synthetic image generator. First, networks are trained on streams of data directly from the generator. In this way, examples are not used more than once during training. Therefore, there is no risk of overfitting and no need to perform additional validation or testing: the training loss for each batch is an unbiased estimate of the generalization loss or risk (2.1). Networks trained in this way should approach the best-possible performance. Hence, the resulting classification performance should be a good indicator of a network's capacity to solve the task. This is also an ideal scenario in which to study the effect of different APR parameters.

In the second experiment, fixed datasets of different size are generated and used to train the networks, emulating more typical real-world scenarios. The networks are validated and tested on separate datasets. This gives insight into the networks' ability to generalize from a finite training set, and the required amount of training data.

4.1.3.1 Network architectures

There are certain difficulties in comparing single-scale pixel CNNs to the developed APRNets. The layers can be nested together in the same order, using the same number of input and output channels, to give the networks equivalent architectures. However, due to the multi-resolution nature of the APR and APRNets, they process the data differently. To allow the an APRNet to make full use of the available information, the convolution kernels used for different resolution levels should be allowed to differ. This does, however, significantly increase the number of parameters of the network. If the APRNet is constrained to use the same kernels across all resolution levels, the number of parameters are equal to the pixel counterpart, but the performance may be suboptimal.

Two baseline network architectures are evaluated in the experiments. These are detailed in Table 4.1. As high resolution information and context should be valuable in distinguishing perturbations from aliasing effects, it is reasonable to give the networks a rather large "direct" receptive field of the original images or APRs. This can be achieved either by successive convolutional layers with smaller kernels, without intermediate pooling, or a single convolutional layer with larger kernel size. In order to save computation time for the APRNets, the latter option is used. The networks are therefore given an initial convolutional layer with a kernel size of 11×11. This is followed by two convolutional layers with 3×3 kernels in one of the architectures, and four such layers in the other. All of these convolutions are followed by batch normalisation, max pooling and rectifier activation functions. Since the number of particles may vary after these layers, the networks are concluded with two layers of 1×1 convolution followed by global average pooling.

Note in Table 4.1 how the level-specific kernels inflate the number of parameters of the APRNets. It could be argued that individual kernels at 5 resolution levels in the first layer is somewhat excessive for this type of image, and likely inflates the parameter count without adding much benefit. Nevertheless, in this way the APRNets are given full freedom. To give more fair direct comparisons, additional

	APR_A	**Pix_A**	**APR_B**	**Pix_B**
conv1	32×11^2 (5) BN, relu pool	32×11^2 BN, relu pool (2^2, str 2)	32×11^2 (5) BN, relu pool	32×11^2 BN, relu pool (2^2, str 2)
conv2	32×3^2 (4) BN, relu pool	32×3^2 BN, relu pool (2^2, str 2)	32×3^2 (4) BN, relu pool	32×3^2 BN, relu pool (2^2, str 2)
conv3	32×3^2 (3) BN, relu pool	32×3^2 BN, relu pool (2^2, str 2)	64×3^2 (3) BN, relu pool	64×3^2 BN, relu pool (2^2, str 2)
conv4	32×1^2 (1) relu	32×1^2 relu	64×3^2 (2) BN, relu pool	64×3^2 BN, relu pool (2^2, str 2)
conv5	6×1^2 (1)	6×1^2	64×3^2 (1) BN, relu pool	64×3^2 BN, relu pool (2^2, str 2)
conv6	-	-	64×1^2 (1) relu	64×1^2 relu
conv7	-	-	6×1^2 (1)	6×1^2
	Global Average Pooling			
#parameters	85088	23520	226592	109728

Table 4.1: The network architectures used in the experiments. Convolutional layers are described as $n \times k^2$, where n specifies the number of output feature maps and k the kernel size. Additionally for the APR networks, the number of resolution-specific kernels are specified in parentheses. Batch normalization is denoted by BN, rectified linear unit activations by relu and max pooling by pool. For the pixel networks, the kernel size and stride of the max pooling are given in parentheses.

results are presented with modified architectures. First, the APRNets may be constrained to use the same kernels across all levels. In the subsequent sections, this is specified by adding the suffix "same" to the network name. Second, as a middle ground between the baseline and "same" APRNets, fewer level-specific kernels can be used. By allowing for two of them in the first layer, and only one in subsequent layers, the number of parameters only increases by 3872. This is specified by the suffix "close". Lastly, instead of modifying the APRNets, the pixel networks can be expanded to use more kernels (channels) in the early layers. In this way, their parameter counts can be increased to match those of the baseline APRNets.

4.1.3.2 Unlimited data availability

Here the results of the first experiment are presented. Each network is trained using 3000 mini-batches, each consisting of 36 images, sampled directly from the generator. The cross entropy loss is used, together with the Adam optimizer. Learning rates are fixed at 0.0005 for all networks. Unless otherwise stated, the relative error threshold E is set to 0.1, corresponding to lossless representation of the APR.

Figure 4.3 shows the learning curves of the small networks, APR_A and Pix_A, as well as the “same” and “close” versions of APR_A that use significantly fewer parameters. The pixel network converges slowly and displays large dips in accuracy. There are several possible explanations for this. Since the batches are rather small, it may simply be that certain batches contain more images that are difficult for the network to classify. It could also be an indication that the learning rate is too high. The APRNets converge significantly faster, and each stage of increased complexity seems to slightly improve both convergence speed and “final” accuracy. Interestingly, there are pronounced dips in accuracy also for the “same” APRNet, but they seem to decrease as more level-specific kernels are allowed.

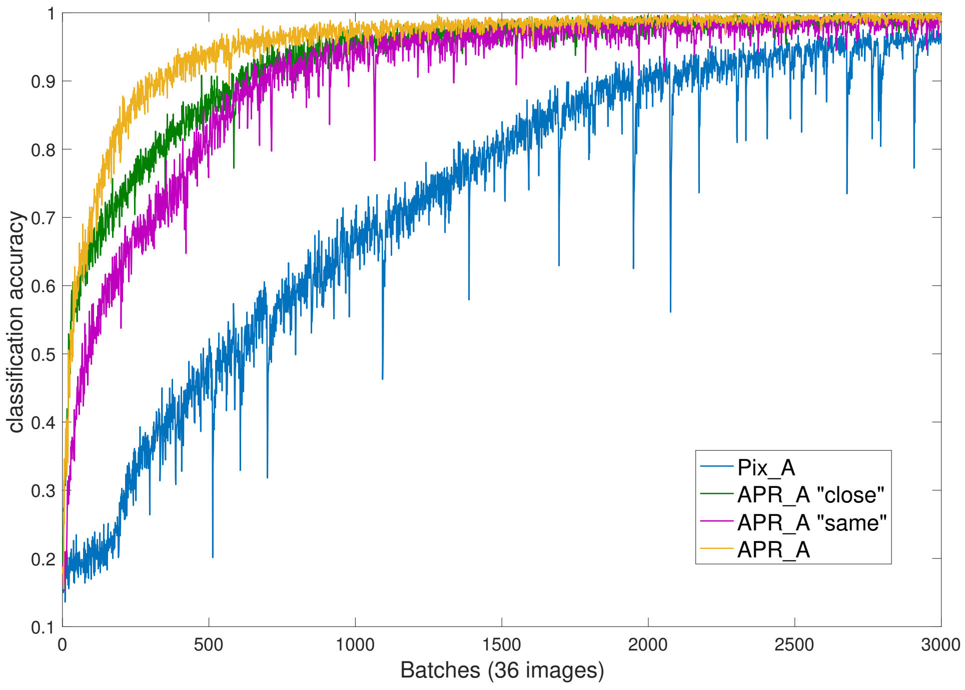


Figure 4.3: Learning results of the small baseline networks, as well as two variations of APR_A whose number of parameters are the same as Pix_A (“same”) and slightly larger (“close”).

Figure 4.4 shows the corresponding results of the larger networks. Increasing the depth of the networks speeds up the convergence in all cases. All of the networks achieve virtually perfect classification after a few hundred batches. Once again, the pixel network and “same” APRNet display pronounced dips in accuracy, whereas the level-adaptive networks do not. Moreover, the level-adaptive APRNets converge significantly faster.

The APR guides the APRNets in the sense that convolutions are applied only where particles exist. In these ideal images, the APR assigns maximum resolution particles only in a band-like region around the object boundary. Therefore, the maximum-level convolutions receive useful (non-constant) input at every position of the kernel. Lower-resolution convolutions should also receive useful input for the

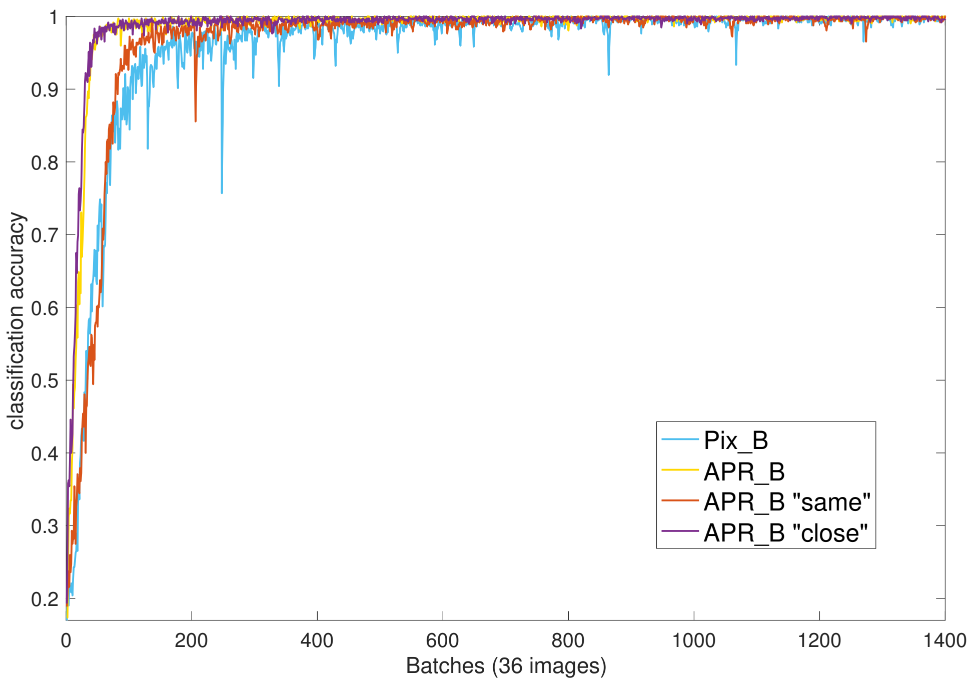


Figure 4.4: Learning results of the larger baseline networks, as well as two variations of APR_B whose number of parameters are the same as Pix_B ("same") and slightly larger ("close").

most part, due to the increased spatial extent of the lower-level kernels. The pixel networks, on the other hand, apply maximum-resoution kernels everywhere in the image. Therefore, a majority of the convolutions are performed in constant regions that convey no useful information. It is possible that the faster convergence of the APRNet is due to this guidance. Irrelevant background regions regions are "filtered out" by the APR, allowing the network to focus the learning around the object boundary. With level-specific kernels, the networks could potentially only focus on this information by giving lower-resolution features less weight. Pixel CNNs, and to some extent the "same" APRNets, must learn to handle the background using the same set of kernels as the feature detection.

4.1.3.3 Impact of the error threshold E

This section presents a simple study of the impact of information loss in the APR formation process on the classification performance of the APRNets. Here, the information loss is controlled by varying the user-set relative error threshold E. Based on the APR performance on the synthetic images, presented previously in Figure 4.2, four values of E that tend to give qualitatively different adaptation have been selected: 0.1, 0.4, 0.6 and 0.8. The typical adaptive properties of the APR for these values are illustrated in Figure 4.5.

For each selected value, APRNets of type APR_B are trained in exactly the same way as in the previous section. That is, using 3000 mini-batches of size 36,

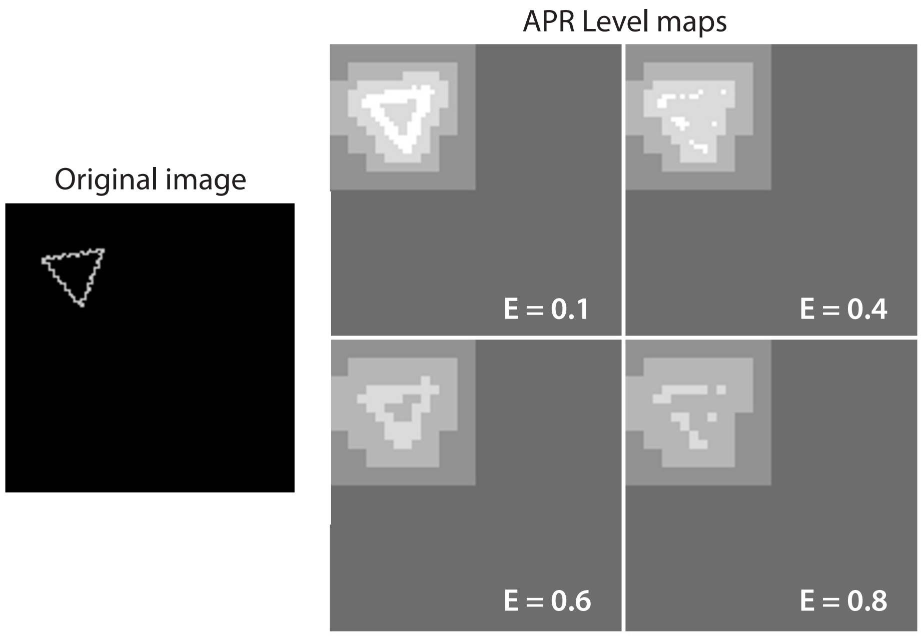


Figure 4.5: Example illustrating the typical adaptation of the APR for the selected values of the error threshold E. White corresponds to pixel-level resolution, whereas darker shades indicate lower resolution. For $E = 0.1$, the object boundary is fully resolved, while for $E = 0.4$ parts of the boundary are downsampled. This pattern is repeated for the larger thresholds. Using $E = 0.6$, the boundary is fully represented at one resolution level below the maximum, while for $E = 0.8$ parts of the boundary are downsampled further.

with the cross-entropy loss, Adam optimizer and a fixed learning rate of 0.0005. In addition, three pixel networks of type Pix_B are trained using pixel images of different resolution. More precisely, one network is trained on the original images, while the remaining two are trained on images downsampled, by means of bilinear interpolation, to size 128×128 and 64×64 pixels, respectively.

To get a rough estimate of the classification performance achieved by the networks, the mean and standard deviation of the error on the last 100 training batches are registered. The results are presented in Table 4.2. Roughly speaking, the networks trained on original images and lossless APRs perform equally. For $E = 0.4$, the APR partially downsamples of the boundary. This is reflected in the classification performance, as the corresponding error lies between those of the full-resolution and once-downsampled pixel networks. Similarly, the APRNet trained with $E = 0.6$ performs roughly equal to the CNN trained on once-downsampled images, while for $E = 0.8$ the error lies between those for once and twice-downsampled pixel images. In other words, the classification error of the APRNets seems to correlate perfectly with the information loss due to the representation. Hence, we may posit that APRNets achieve similar classification performance to regular CNNs applied to the reconstructed images. However, this claim does need further verification.

Data type	image size / E	Classification error
APR	$E = 0.1$	0.0013 ± 0.0015
Pixels	256×256	0.0016 ± 0.0028
APR	$E = 0.4$	0.0071 ± 0.0031
APR	$E = 0.6$	0.0102 ± 0.0048
Pixels	128×128	0.0116 ± 0.0040
APR	$E = 0.8$	0.0261 ± 0.0067
Pixels	64×64	0.0770 ± 0.0123

Table 4.2: Classification performance of the network APR_B, trained using four different values of the reconstruction error threshold E, and Pix_B trained using full-size (256×256), once-downsampled (128×128) and twice-downsampled (64×64) images. The classification errors and standard deviations are computed from the last 100 training batches, each consisting of 36 images.

4.1.3.4 Finite datasets

The ideal scenario in the experiments above, being able to sample the generating distribution indefinitely, is not common in practice. Instead, one typially only has a finite set of data, divided into subsets for training, validation and testing. Here, this scenario is emulated by generating finite sets of synthetic images. Several networks are trained on datasets of different size. Hence, the performance of the networks should give insight into their ability to generalize from finite datasets, as well as the amount of training data required to reach a certain accuracy.

To this end, seven training datasets of different size have been generated in such a way that the smaller datasets are subsets of the larger ones. These datasets contain 20, 50, 100, 200, 500, 1000 and 2000 labeled images from each class. Independent datasets are used for validation and testing, each containing 500 images per class. These are generated using the same parameter settings, so that the training, validation and test datasets consist of independent, identically distributed examples. However, in practice it may happen that the training dataset only represents a subset of the distribution for which the network is employed. This scenario is emulated by testing the models on yet another dataset, consisting of 500 images per class generated with slightly different parameters[1].

For each of the training datasets, networks of type APR_B and Pix_B are trained. Since overfitting is likely to be a problem, especially with the smaller training datasets, several regularization techniques are employed during training. Spatial dropout is used after the last max pooling layer, with a probability $p_{drop} = 0.3$ to zero out any given feature map. Between the 1×1 convolution layers, regular dropout is used to zero out individual neurons with probability $p_{drop} = 0.5$. Finally, weight decay (L_2 regularization) is employed with parameter $\lambda = 1e - 4$. During training, the networks are evaluated on the validation dataset in regular intervals, and the models with the minimum validation loss saved. These models are then evaluated on the test datasets, the results of which are shown in Figure 4.6.

[1]The parameters are modified to be "more extreme", so that the training data represent a subset of the new distribution.

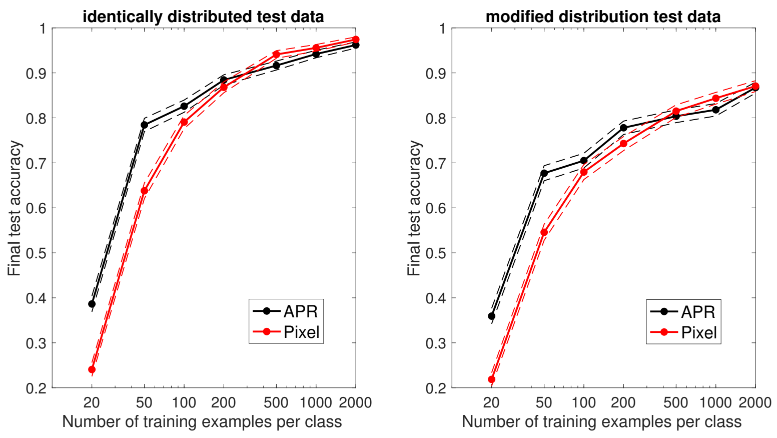


Figure 4.6: Classification accuracy on the test datasets (500 images per class) for APR and pixel networks trained on fixed datasets of different size. One test dataset is distributed identically to the training data (left), while the other is sampled from a modified distribution (right). Approximate 95% confidence intervals for the accuracy measurements are shown by dashed lines.

Interestingly, the APRNets trained on very small datasets perform significantly better than their pixel counterpart. Once again this could be attributed to the higher information density of the APR compared to the pixel representation. However, there is also a significant amount of randomness involved in the procedure above. Further experiments are required to make any definitive claims. When trained on larger datasets, the APR and pixel networks display similar performance, with the pixel networks performing slightly better. Here the confidence intervals are overlapping, so it is not clear whether or not the differences are statistically significant.

4.2 Natural image benchmark

The synthetic images used in the previous section are, in a sense, optimal for the APR due to their extreme sparsity and lack of noise. In this section, APRNets are evaluated on the classical task of distinguishing natural images of cats and dogs, which are much less ideal for the APR. First, a brief description of the dataset is given. This is followed by a summary of the APR's performance on the images, details on the benchmarked network architectures, and finally the results.

4.2.1 Dogs vs. Cats dataset

The dataset used is the set of training images from the Dogs vs. Cats competition on Kaggle[2]. It consists of 25000 labeled images of cats and dogs, distributed evenly between the classes. The images are a subset of the Asirra database [47], which contains over three million manually labeled photographs.

There is tremendous variation in quality and resolution of the images in the dataset. Processing batches of different-sized images is somewhat problematic in

[2]`https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition`

practice, as they cannot be stacked into regular arrays. This could be solved by zero-padding the smaller images to match the size of the largest image in the batch. However, since the image sizes vary over an order of magnitude in the dataset, zero-padding would result in large amounts of redundant computation. For this reason, the images are resized to 256×256 pixels by means of bilinear interpolation prior to processing or transformation to the APR.

4.2.2 APR performance

Although the current implementation of the APR does not support multi-channel images, a simple workaround can be employed. To convert color images, they are first converted to grayscale to allow computation of the APR, after which each color channel is resampled at the particle locations. It should be noted that the reconstruction condition is only guaranteed for the grayscale image, and not the individual channels. However, the procedure seems to work well in practice.

Figure 4.7 summarizes the performance on a set of 1000 randomly sampled images. Clearly, the computational benefits of the APR are limited in comparison to the sparse synthetic images. For the most conservative error threshold, $E = 0.2$, the computational ratio is roughly 2 on average. It then approximately doubles for each successive value. Looking at the average errors, the largest step occurs between $E = 0.2$ and $E = 0.4$. Further increasing the threshold seems to slightly increase both the mean and variance of the error measurements.

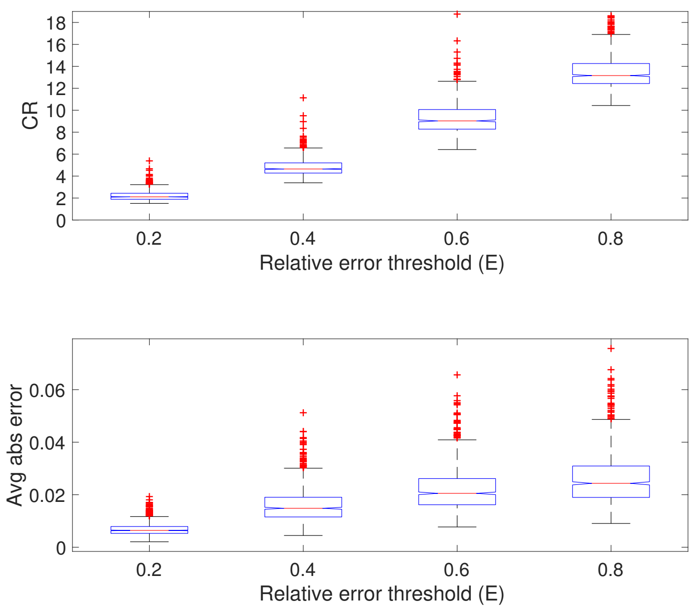
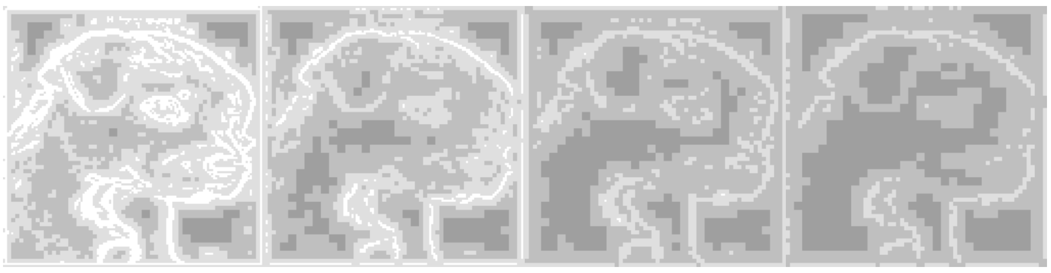


Figure 4.7: APR performance on a set of 1000 randomly sampled images from the cats and dogs dataset, for different values of the error threshold E. The images are resized to 256×256 pixels prior to transformation. (Top) Benefit of the APR in terms of the computational ratio. (Bottom) Average absolute reconstruction error, taken over all pixels.

Figure 4.8 illustrates the APR adaptation for an example image. In this case, the APR fully resolves most of the edges for $E = 0.2$, which is reflected in the smoothness of the reconstructed image. Higher values of E lead to some loss of fine detail, in particular around the ear, nose and eye, where the contrast is not as high as between the contours of the dog and the background. This is also reflected in the reconstructions, by the increasingly pronounced pixelation.



(a) $E = 0.2$ **(b)** $E = 0.4$ **(c)** $E = 0.6$ **(d)** $E = 0.8$

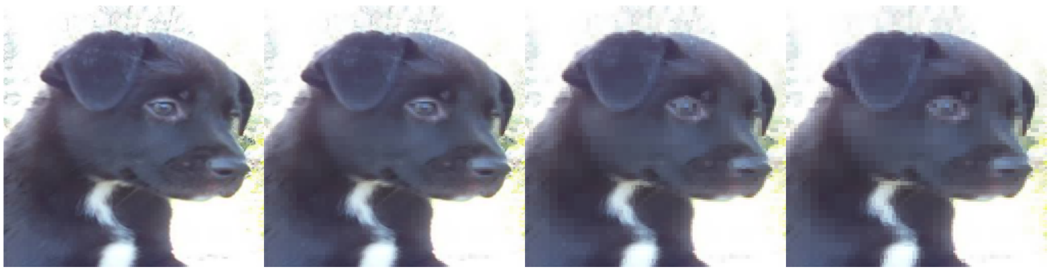


Figure 4.8: Example of APR performance for different values of the reconstruction error threshold E. The top row shows the adaptation by mapping particle resolution levels to pixels (brighter means higher resolution), whereas the bottom row shows the corresponding reconstructed images.

4.2.3 Network architectures

Three network architectures are evaluated on the task. These are detailed in Table 4.3. Notably, the APRNet is here given an auxiliary input channel, encoding the resolution level of the particles, normalized to the interval $[0, 1]$. Furthermore, level-specific kernels are used in the APRNet, leading to an increased number of parameters. To account for this and allow for more fair comparisons, an expanded version of the pixel network is evaluated. Judging the potential of the networks solely by number of parameters, the APRNet can be expected to place somewhere between the pixel networks. However, this is of course an overly simplistic view, as the performance of any neural network depends on a large number of additional variables.

	Pixels	**APR**	**Pixels (expanded)**
input channels	RGB	RGB + level	RGB
conv1	32×3^2 BN, relu pool (2^2, str 2)	32×3^2 (4) BN, relu pool	128×3^2 BN, relu pool (2^2, str 2)
conv2	64×3^2 BN, relu pool (2^2, str 2)	64×3^2 (3) BN, relu pool	128×3^2 BN, relu pool (2^2, str 2)
conv3	128×3^2 BN, relu pool (2^2, str 2)	128×3^2 (2) BN, relu pool	128×3^2 BN, relu pool (2^2, str 2)
conv4	128×3^2 BN, relu pool (2^2, str 2)	128×3^2 (1) BN, relu pool	128×3^2 BN, relu pool (2^2, str 2)
conv5	128×1^2 relu	128×1^2 (1) relu	128×1^2 relu
conv6	2×1^2	2×1^2 (1)	2×1^2
	Global Average Pooling		
#parameters	257473	371809	462977

Table 4.3: Network architectures for cats vs dogs classification. Convolutional layers are described as $n \times k^2$, where n specifies the number of output feature maps and k the kernel size. Additionally for the APR networks, the number of resolution-specific kernels are specified in parentheses. Batch normalization is denoted by BN, rectified linear unit activations by relu and max pooling by pool. For the pixel networks, the kernel size and stride of the max pooling are given in parentheses. The APR network receives input with four channels, encoding RGB color intensities as well as the particle levels.

4.2.4 Classification performance

To evaluate the networks, the dataset is randomly divided into 24000 images for training and 1000 for validation. Data augmentation is employed during training, where the images undergo the following transformations:

- resize to 256×256 pixels (bilinear interpolation)
- random choice of:
  - random rotation (max 30 degrees)
  - color jitter (brightness and contrast scaled by a random factor $r \in [0.8, 1.2]$)
  - random translation ($\pm 5\%$) and scaling ($\pm 10\%$)
- random horizontal flip ($p = 0.5$)

Images used for validation are resized to 256×256, but not augmented. For the APRNet, the input APRs are computed from the transformed images. All networks are trained using a mini-batch size of 24 and a fixed learning rate of $1e-4$. After each pass through the training dataset, or "epoch", the networks are evaluated on the validation dataset.

Figure 4.9 shows the validation accuracy during training of the networks. It is

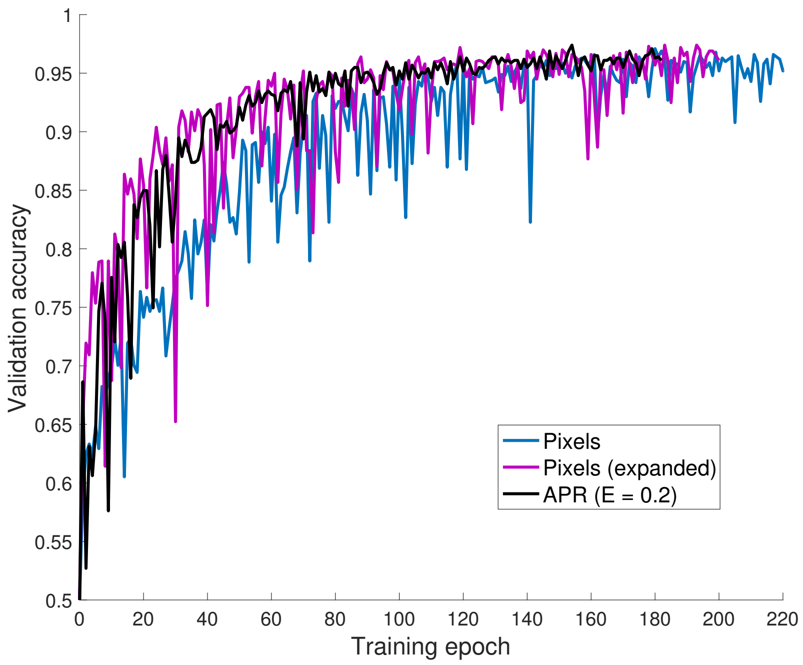


Figure 4.9: Progression of the validation accuracy during training of the APRNet and two pixel networks on the cats and dogs dataset.

seen that all networks converge to roughly the same accuracy. The APRNet and expanded pixel network converge faster than the smaller pixel network. Once again, both pixel networks display significant dips in accuracy, whereas the APRNet is more stable. As previously mentioned, this could indicate that the learning rate is too high for the pixel networks. However, since all networks are trained using a fixed learning rate, it begs the question of whether APRNets allow for higher learning rates in general.

Quantitative measures of the network performances are given in Table 4.4. This includes three more APRNets, trained using different values of the error threshold E. For each network, the highest achieved validation accuracy is reported. It should be noted that these values are biased estimates of the true generalization accuracy. Better comparisons could be made by evaluating the selected models on a held-out test dataset. Nevertheless, taking into account that the exact values ought to be taken with a grain of salt, they do give some insight into the performance of the networks. It may be concluded that the pixel networks and the APRNet trained using $E = 0.2$ achieve similar performance. APRNets trained with higher values of E suffer slightly in terms of accuracy. Although, interestingly, the loss of accuracy when increasing E from 0.4 to 0.8 seems to be only marginal, despite the pronounced difference in quality displayed in Figures 4.7 and 4.8. For $E = 0.8$, the details in the images are essentially represented at once resolution level lower than for $E = 0.4$.

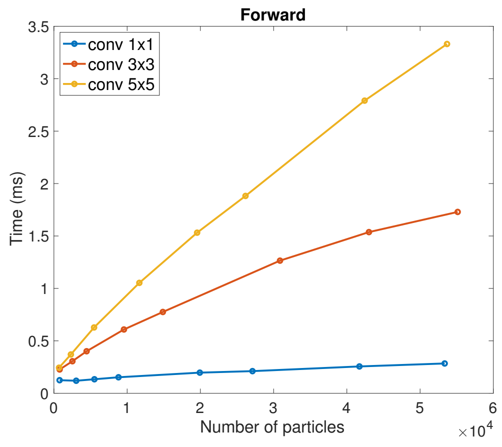
Network	Top validation accuracy
Pixels	0.9709
Pixels (expanded)	0.9739
APR ($E = 0.2$)	0.9739
APR ($E = 0.4$)	0.9589
APR ($E = 0.6$)	0.9579
APR ($E = 0.8$)	0.9549

Table 4.4: Highest achieved validation accuracies of the pixel networks, as well as the APRNet trained using different values of the reconstruction error threshold E.

4.3 Computational scalings of APRNet operations

To assess the computational scalings of the layer operations, suppose that an input APR with N_p particles and C channels is given. The pooling operation (Algorithm 1) is easily seen to have a worst-case computational cost of $\mathcal{O}(CN_p)$. For the convolution operation (Algorithm 3), suppose that D kernels of size $C \times k \times k$ are applied, resulting in an output APR with N_p particles and D channels. Each output value is computed using $Ck^2 + 1$ multiply-add operations, where the one accounts for addition of bias terms. Thus, computing all of the DN_p output values requires $D(Ck^2 + 1)N_p$ floating-point operations. However, the APRTree must be filled for each of the C input channels. This requires $4CN_t$ operations, where $N_t < N_p$ is the number of tree nodes. Hence, in big O notation, the computational complexity of the forward convolution is $\mathcal{O}(CDk^2N_p)$. In the backward pass of the convolution operation (Algorithm 4), the DN_p output-gradient values are propagated to the inputs and parameters. For each output position, Ck^2 contributions go to the inputs and another $Ck^2 + 1$ go to the parameters (Ck^2 to the weights and 1 to the bias term). Thus, the operation requires $D(2Ck^2 + 1)N_p$ operations. In addition, the APRTree has to be filled, and the gradient tree unloaded, for each input channel. This results in an addition of $8CN_t$ operations. Once again the cost grows asymptotically as $\mathcal{O}(CDk^2N_p)$, and the actual cost is roughly twice the cost of the forward operation.

Now, the analysis above does not give any information about the real-time consumption of the algorithms. To assess this, synthetic images containing different numbers of objects are generated. As the number of objects increase, the sparsity decreases and the number of particles required to represent the image increases. The APRs are then passed repeatedly to the convolution and pooling operations to measure the average time required for each operation. Figures 4.10 and 4.11 illustrate some measurements obtained in this way. The timings are recorded for a single APR, one input channel and one output channel. Due to the hard-coded OpenMP batch parallelism in the current implementation, this means that all processing is done by a single thread. The presented measurements are obtained using a MacBook Pro Retina (2013), equipped with a 2.3GHz Intel Core i7 processor. It is clearly seen that the computation time of each operation scales linearly with the number of particles. Moreover, the time required by the backward convolution passes are indeed almost exactly two times the forward computation time in each case.



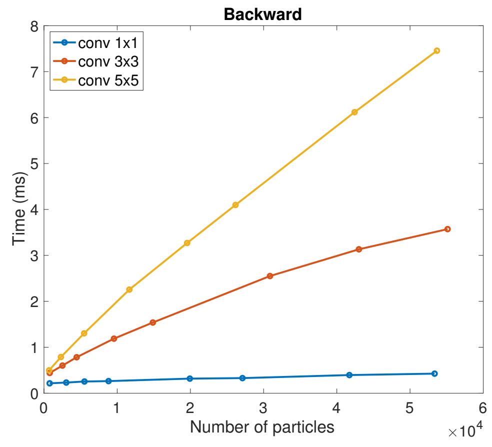


Figure 4.10: Measurements of the real time required by the forward and backward passes of the convolution operations. Each measurement is the average time of 5000 operation calls, recorded after a burn-in of 500 calls.

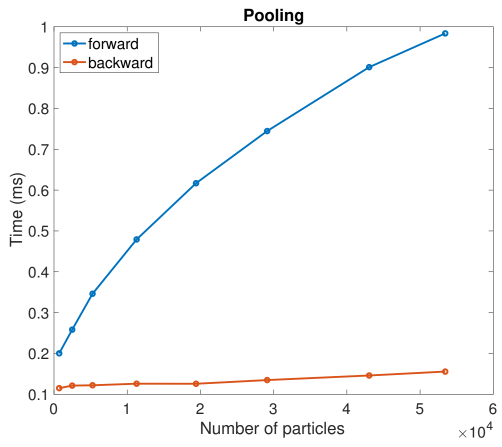


Figure 4.11: Measurements of the real time required by the forward and backward pass of the max pooling operation. Each measurement is the average time of 10000 operation calls, recorded after a burn-in of 1000 calls.

In light of these results, convolutions and pooling on the APR have the potential to be faster than the corresponding pixel operations by a factor equal to the computational ratio. In reality this is likely not achievable, due to the overhead work required by the APR operations. However, for large enough computational ratios, significant reduction in real processing time can be achieved.

4.3.1 Memory benefits of APRNets

Here the potential reductions in memory usage of APRNets compared to pixel CNNs are discussed. In the application of any given neural network model, the inputs are passed through successive layer operations to compute an output. During training, the inputs received by each layer in the forward pass must be stored in memory for the backward pass. Thus, at the very end of the forward pass, the network input as well as every intermediate feature map are stored in memory. This, together with the parameters of the network and potentially some additional workspace buffers, make up the memory footprint of the network.

In order to assess the potential memory benefits of APRNets, the memory footprint of input and intermediate feature maps are computed for an example network architecture. The cost of storing the network itself, as well as temporary memory usage of individual layer operations, are ignored. For the input to the network and feature maps before any pooling layers, the APR reduces the memory cost by a factor equal to the computational ratio (CR). Each successive pooling operation makes the APR more dense. Thus, the CR of pooled feature maps is smaller than the input CR. Because of this, the exact memory footprint of the APR depends on the adaptation to the specific input image. This complicates the analytical study of the memory footprint of APRNets, but it can be studied empirically by computing the feature map sizes for specific inputs.

As an example of the potential reductions in memory usage of APRNets, the feature extractor part of the VGG16 network described in Figure 2.6 is considered. Since this part is fully convolutional, it can accept inputs of any size. To compute the memory footprint of the network, the sizes of all the layer inputs are accumulated. This is done for synthetic images of size 512×512 pixels containing increasing numbers of objects. Figure 4.12 shows the memory required to store the feature maps as APRs relative to the pixel representation. In this case, the increased representation

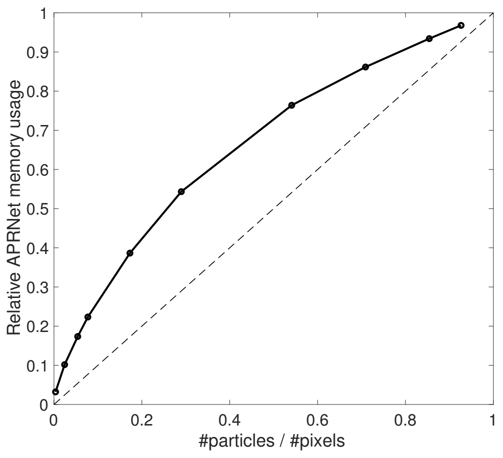


Figure 4.12: Example of the benefit of APRNet to the memory cost of storing intermediate feature maps. Feature map sizes are computed for increasingly dense synthetic images of size 512×512 pixels to obtain the memory usage. The x-axis shows the reduction in network input size and the y-axis shows the memory cost of the APR feature maps relative to the pixel versions.

density combined with the greatly increased number of channels in the later layers causes the scaling to differ quite significantly from linearity[3]. Nevertheless, it is clear that significant reductions in memory usage can be achieved for large enough

[3]The relationship can be brought closer to linearity by altering the network architecture. In this way the memory benefits can be controlled by the user, to a certain degree

computational ratios. It should also be noted that the actual memory benefit will increase significantly when a third spatial dimension is included, due to the increased contribution from the high-resolution images to the total memory footprint.

5 Conclusions and future work

5.1 Conclusions

In this thesis project, convolutional neural networks (CNNs) for the Adaptive Particle Representation (APR) have been developed, implemented and evaluated. This has been achieved by extending the operations of convolution and pooling layers to the APR, and we call the resulting network type APRNet. The developed operations share several similarities with existing processing methods for image pyramids, but differ in that computations are applied in a sparse manner to select spatial locations defined by the APR. In this way, the APR guides the network operations and focuses the feature extraction to regions of high information content, such as edges.

The APRNet operations have been implemented in C++, using multithreading by OpenMP to speed up computations. In order to enable seamless building and application of APRNets, the functionality has further been wrapped to Python and implemented as custom, high-level PyTorch modules. As a result, APRNets can be built and trained in PyTorch similarly to traditional CNNs, with only marginal added complexity for the user.

To evaluate the performance of APRNets, several networks have been built and trained on tasks of image classification using both synthetic and natural images. The results clearly show that APRNets are able to learn and achieve classification performances similar to pixel CNNs with equivalent architectures. In addition, several interesting observations about the learning characteristics of APRNets can be made from the results:

- When training equivalent APRNet and CNN architectures with a fixed learning rate, the APRNets display a more stable progression of validation accuracy. This seems to indicate that processing on the APR has a regularizing effect, and can be attributed to the fact that the feature extracting filters receive fewer inputs, with increased proportions of meaningful signals.
- In the task of synthetic image classification, the APRNets seem to converge significantly faster than equivalent pixel CNNs. This is most likely a consequence of the above bullet point. For natural images, this is not as apparent. However, the regularizing effects can be expected to be much less pronounced in this case due to the high density of the particle representation.
- A third observation, further supporting the hypothesis that the APR has a regularizing effect on the learning, is that the APRNets are able to generalize significantly better from very small datasets of synthetic images.

Finally, the computational costs of the APRNet layer operations have been shown, both empirically and analytically, to scale linearly with the number of particles rather than the original image size. Similarly, the memory cost of intermediate feature maps scale with the number of particles. Therefore, the APRNet has the potential to significantly reduce computational and memory costs in the processing of large and sparse images. These reductions are most significant in the early layers of the network, and diminish with each pooling layer as the density of the APR is increased.

5.2 Future work

The presented work has served as a proof of concept of the potential of APRNets. Comparisons have been made between APRNets and single-scale CNNs. However, due to the multi-resolution nature of the APR, these network types process data in fundamentally different ways. By instead considering multi-resolution CNNs for image pyramids, which have a clear connection to APRNets, better direct comparisons can be made. In addition, several improvements and extensions can be made to increase the practical usefulness of APRNets:

- Perhaps most importantly, the implementation of the operations can be optimized to increase the computational speed. We expect that tremendous speedups can be achieved by utilizing graphics processing units (GPU) to perform the computations. Several of the elements necessary to implement the operations for GPU computing already exist as part of previously implemented pipelines. It is therefore expected that full GPU implementations of the convolution and pooling operations are possible.
- Since the primary target application of the APR concerns large, 3D fluorescence microscopy images, the operations should also be extended to three spatial dimensions. This can be done trivially for the current implementation. However, processing such data in reasonable time would require significant improvements to the computational efficiency of the implementation.
- To enable the use of APRNets in a wider range of learning tasks, additional layer operations are required. For instance, modern CNNs for image segmentation [19, 2] and restoration [1] typically employ learnable upsampling operations. We expect that these, too, can be extended to the APR similarly to the convolution and pooling operations treated here.
- Finally, we envision that hybrids between APRNets and CNNs can be useful for many reasons. For instance, neural networks mapping APRs to images could be used to optimally reconstruct and restore pixel images. Moreover, the APR becomes increasingly dense and approaches the pixel representation as it undergoes successive pooling operations. Hence, real-time computational benefits could be made possible by using the APR in the early layers, and switching to the pixel representation in the later layers to make use of more efficient algorithms.

Bibliography

[1] M. Weigert, U. Schmidt, T. Boothe, A. Müller, A. Dibrov, A. Jain, B. Wilhelm, D. Schmidt, C. Broaddus, S. Culley, *et al.*, “Content-aware image restoration: pushing the limits of fluorescence microscopy,” *Nature methods*, vol. 15, no. 12, p. 1090, 2018.

[2] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241, Springer, 2015.

[3] M. Weigert, L. Royer, F. Jug, and G. Myers, “Isotropic reconstruction of 3d fluorescence microscopy images using convolutional neural networks,” in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 126–134, Springer, 2017.

[4] U. Schmidt, M. Weigert, C. Broaddus, and G. Myers, “Cell detection with star-convex polygons,” in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 265–273, Springer, 2018.

[5] E. G. Reynaud, J. Peychl, J. Huisken, and P. Tomancak, “Guide to light-sheet microscopy for adventurous biologists,” *Nature methods*, vol. 12, no. 1, p. 30, 2014.

[6] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, “vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design,” in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 18, IEEE Press, 2016.

[7] B. L. Cheeseman, U. Günther, K. Gonciarz, M. Susik, and I. F. Sbalzarini, “Adaptive particle representation of fluorescence microscopy images,” *Nature communications*, vol. 9, no. 1, p. 5160, 2018.

[8] A. K. Jain and S. Z. Li, *Handbook of face recognition.* Springer, 2011.

[9] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, “Dermatologist-level classification of skin cancer with deep neural networks,” *Nature*, vol. 542, no. 7639, p. 115, 2017.

[10] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, p. 436, 2015.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[12] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (G. Gordon, D. Dunson, and M. Dudík, eds.), vol. 15 of *Proceedings of Machine Learning Research*, (Fort Lauderdale, FL, USA), pp. 315–323, PMLR, 11–13 Apr 2011.

[13] D. W. Ruck, S. K. Rogers, M. Kabrisky, M. E. Oxley, and B. W. Suter, "The multilayer perceptron as an approximation to a bayes optimal discriminant function," *IEEE Transactions on Neural Networks*, vol. 1, no. 4, pp. 296–298, 1990.

[14] U. Orhan, M. Hekim, and M. Ozer, "Eeg signals classification using the k-means clustering and a multilayer perceptron neural network model," *Expert Systems with Applications*, vol. 38, no. 10, pp. 13475–13481, 2011.

[15] S. Mishra, R. Yadav, and R. Singh, "A survey on applications of multi layer perceptron neural networks in doa estimation for smart antennas," *International Journal of Computer Applications*, vol. 83, no. 17, 2013.

[16] M. H. Esfe, M. Afrand, S. Wongwises, A. Naderi, A. Asadi, S. Rostami, and M. Akbari, "Applications of feedforward multilayer perceptron artificial neural networks and empirical correlation for prediction of thermal conductivity of mg (oh) 2–eg using experimental data," *International Communications in Heat and Mass Transfer*, vol. 67, pp. 46–50, 2015.

[17] W. Rawat and Z. Wang, "Deep convolutional neural networks for image classification: A comprehensive review," *Neural computation*, vol. 29, no. 9, pp. 2352–2449, 2017.

[18] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Advances in neural information processing systems*, pp. 649–657, 2015.

[19] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[20] K. Zhang, W. Zuo, Y. Chen, D. Meng, and L. Zhang, "Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising," *IEEE Transactions on Image Processing*, vol. 26, no. 7, pp. 3142–3155, 2017.

[21] C. Dong, C. C. Loy, K. He, and X. Tang, "Image super-resolution using deep convolutional networks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 2, pp. 295–307, 2016.

[22] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*, pp. 818–833, Springer, 2014.

[23] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[24] I. S. A. Krizhevsky and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[25] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[27] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," in *European conference on computer vision*, pp. 346–361, Springer, 2014.

[29] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *arXiv preprint arXiv:1804.07612*, 2018.

[30] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[31] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[32] A. C. Oates, N. Gorfinkiel, M. Gonzalez-Gaitan, and C.-P. Heisenberg, "Quantitative approaches in developmental biology," *Nature Reviews Genetics*, vol. 10, no. 8, p. 517, 2009.

[33] S. G. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 7, pp. 674–693, 1989.

[34] E. H. Adelson, C. H. Anderson, J. R. Bergen, P. J. Burt, and J. M. Ogden, "Pyramid methods in image processing," *RCA engineer*, vol. 29, no. 6, pp. 33–41, 1984.

[35] B. Cheeseman, K. Gonciarz, U. Günther, J. Jonsson, M. Emmenlauer, and M. Susik, "cheesema/libapr: Initial release v1.1," Sept. 2018.

[36] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pp. 233–244, ACM, 2009.

[37] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[38] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1915–1929, 2013.

[39] D. Yoo, S. Park, J.-Y. Lee, and I. So Kweon, "Multi-scale pyramid pooling for deep convolutional representation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 71–80, 2015.

[40] G. Nam, H. Choi, J. Cho, and I.-J. Kim, "Psi-cnn: A pyramid-based scale-invariant cnn architecture for face recognition robust to various image resolutions," *Applied Sciences*, vol. 8, no. 9, p. 1561, 2018.

[41] S. Yang and D. Ramanan, "Multi-scale recognition with dag-cnns," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1215–1223, 2015.

[42] G. Riegler, A. Osman Ulusoy, and A. Geiger, "Octnet: Learning deep 3d representations at high resolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3577–3586, 2017.

[43] B. Graham, "Spatially-sparse convolutional neural networks," *arXiv preprint arXiv:1409.6070*, 2014.

[44] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *Computing in Science & Engineering*, no. 1, pp. 46–55, 1998.

[45] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11 – seamless operability between c++11 and python," 2017. https://github.com/pybind/pybind11.

[46] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[47] J. Elson, J. J. Douceur, J. Howell, and J. Saul, "Asirra: a captcha that exploits interest-aligned manual image categorization," 2007.