Mathematisch-Naturwissenschaftliche Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

Signals processing in the Graph Convolutional Networks

by

Andrii Lishchyshyn

andrej.lischishin@uni-bonn.de

Born 25th June 1997 in Kiev, Ukraine

Supervisor: Prof. Dr. Michael Griebel

Second supervisor: Dr. Bastian Bohn

Bachelor's Thesis Mathematics

in the

Institute for Numerical Simulation

Research Group M. Griebel

November 7, 2018

“...”

Sensei Six

Zusammenfassung

Während des laufenden Jahrzehnts hat Deep Learning in verschiedenen Bereichen, von Computer Vision und Bildanalyse bis hin zu Spracherkennung und natürlicher Sprachverarbeitung, ein herausragendes Leistungsniveau erreicht. Heutzutage ist Deep Learning eine eigenständige Technologie, die in kommerziellen Anwendungen verwendet wird, einschließlich Siri, Face ID in IPhone (Apple), Googletext-Übersetzung, Betrugserkennung (Sentinel Protocol) und vielen anderen. In dieser Arbeit liegt der Schwerpunkt auf einem Teilbereich des Deep Learning, insbesondere der Convolutional Neural Networks (CNNs). CNNs liefern derzeit eine State-of-the-Art-Leistung für eine Vielzahl von Computer-Vision-Aufgaben. Die Forschung konzentrierte sich hauptsächlich auf den Umgang mit 2D- oder 3D-euklidisch strukturierten Daten wie Bildern, Videos oder akustischen Signalen. Während die Erweiterung von 2D auf 3D euklidisch strukturierte Daten einfach ist, da Daten immer noch eine Gitterstruktur aufweisen, ist die Erweiterung auf andere Arten von Daten wie 3D-Form-Mashes, Graphen sozialer Netzwerke oder IoT-Graphen nicht sehr klar. In diesem Zusammenhang werden wir eine neue Graph-Faltungsnetzwerkarchitektur untersuchen, die auf einer generischen Formulierung basiert, die die 1-zu-1-Entsprechung zwischen Filtergewichten und Datenelementen ändert. Die Hauptnovalität dieses Ansatzes besteht darin, dass die Form des Filters eine Funktion der Merkmale in der vorherigen Netzwerkschicht ist, die als ein integraler Teil des neuronalen Netzwerks gelernt wird. Dieser Ansatz wurde in [1] vorgeschlagen.

Mathematisch-Naturwissenschaftliche Fakultät der Rheinischen
Friedrich-Wilhelms-Universität Bonn

Abstract

Institute for Numerical Simulation

Bachelor Thesis

by Andrii Lishchyshyn
andrej.lischishin@uni-bonn.de
Born 25th June 1997 in Kiev, Ukraine

Over the current decade, deep learning has achieved an outstanding performance level in different fields, from computer vision and image analysis to speech recognition and natural language processing. Nowadays, deep learning is a self-standing technology that is used in commercial applications, including Siri, Face ID in IPhone (Apple), Google text translation, fraud detection (Sentinel Protocol) and many others. In this work, the main focus is on a subarea of deep learning, in particular convolutional neural networks (CNNs). CNNs currently produce state-of-the-art performance on a wide variety of computer vision tasks. Mostly, research was focused on dealing with 2D or 3D Euclidean-structured data, such as images, videos or acoustic signals. While the extension from 2D to 3D Euclidean-structured data is straightforward because data still has a grid structure, extension to other types of data such as 3D shape mashes, social networks graphs or Internet of Things (IoT) graphs is not very clear. In this regard, we will examine a new graph-convolutional network architecture that builds on a generic formulation which changes the 1-to-1 correspondence between filter weights and data elements. The main novelty of this approach is that the shape of the filter is a function of the features in the previous network layer, which is learned as an integral part of the neural network. This approach was proposed in [1].

Acknowledgements

I wish to express my thanks to Prof. Dr. Michael Griebel and Dr. Bastian Bohn for an interesting research topic, fruitful discussions and constant support....

Contents

Chapter 1

Introduction

Convolutional neural networks [2] introduce an efficient architecture to extract meaningful patterns in high-dimensional and large-scale datasets. CNNs are able to learn local structures and compose them into multi-scale patterns. This has led to dramatic improvement in video, image and sound recognition tasks. A basic CNN consists of an input layer, multiple hidden layers and an output layer. At first, this looks just like a simple neural network, but the difference is in the functioning of hidden layers. The hidden layers of a CNN typically consist of convolutional layers, pooling layers, fully connected layers and normalization layers. To be precise, CNNs extract the local stationarity property of the input data by revealing local features that are shared across the data domain. These features are identified with localized convolutional filters or kernels, which are learned from the data during iterative learning of the network. Convolutional Networks combine three architectural ideas to ensure some degree of shift, scale and distortion invariance: local receptive fields, shared weights and sub-sampling.

3D shape models, user data on social networks, genetic data on biological regulatory networks or text documents on word embeddings are important examples of data lying in irregular or non-Euclidean domains that can be represented with graphs. Graphs can encode complex geometric structures and there is a variety of mathematical tools that can be used for studying the graphs.

The non-Euclidean nature of such data implies that there are no such familiar properties as global parametrization, vector space structure or shift-invariance. Therefore, basic operations like convolution that are taken for granted in an Euclidean case are even not well-defined on non-Euclidean domains. The purpose of this work is to study one specific architecture of convolutional neural network for graph-data.

Previous works on deep neural networks over graph-structures data can be divided into spectral [3–6] and spatial approaches. Spectral filtering approaches rely on the eigen-decomposition of the graph Laplacian that enables convolutions over graphs. It is useful for problems with fixed graph representation, but non-local eigen-decomposition is unstable when applied across different graphs, which makes the generalization across different graphs rather difficult. In contrast, spatial approaches [7–9] provide filter localization via the finite size of the kernel. However, although graph convolution in the spatial domain is conceivable, it faces the challenge of matching local neighbourhoods, as pointed out in [4].

In this work, we will look at one specific graph convolutional neural network based on local filtering. In this approach, the mapping between the filter weights and the nodes in a neighborhood of the graph are learned as an integral part of the network. Moreover, the mapping is a function of the features in the preceding layer of the network, rather than being based on manually defined local coordinates on the graph.

1.1 Neural Networks

We will start with a short introduction to the Neural Networks in general. The first steps in the area of neural networks were already made in the 1940s. An artificial neural network is a computing system vaguely inspired by the biological neural networks that constitute animal brains. A neural network itself is not an algorithm, but rather a framework for many different machine learning algorithms to work together and process complex data inputs. Such systems "learn" to perform tasks by considering examples, generally without being programmed with any task-specific rules.
A basic neural network has a clear structure:

- **Input layer**: No computation is done within this layer. It serves as a place where data is fed to the network in order to produce some output.
- **Hidden layer**: Hidden layers are the essence of the network. They are places where intermediate processing or computation is done, they perform computations and then transfer the weights (signals of information) from the input layer to the following layer (another hidden layer or the output layer).
- **Output layer**: Here we finally use some specific activation function that maps to the desired output format.
- **Connections and weights**: The network consists of connections, each connection transferring the output from one layer to the input of the other layer.

- **Activation function**: The activation function of a layer defines the output of each node in that layer given an input or set of inputs.
- **Learning rule**: The learning rule is a rule or an algorithm which modifies the parameters of the neural network, in order for a given input to the network to produce a favored output. During this learning process, weights are typically modified in a way that increases the accuracy of the network's prediction.

A key trigger for renewed interest in neural networks and learning was Werbo's (1975) backpropagation algorithm that effectively solved the exclusive-or problem by making the training of multi-layer networks efficient. Backpropagation is used in the learning process and distributes the error term back up through the layers, by modifying the weights at each node.
There are many classes of neural networks and these classes also have sub-classes. Classes are usually formed by a type of layers or in general by the architecture of the network and also by a task or data which are to be solved or is used.
In this work we are interested in specific architecture and a specific dataset, namely Convolutional Neural Network on Graphs, which means that data is given as a graph.

1.2 Convolutional Neural Networks

In this section, we will look at an example of a convolutional neural network applied to the digit recognition problem. We will follow one of the earliest examples in the field, which is LeNet-5, first introduced in [2].

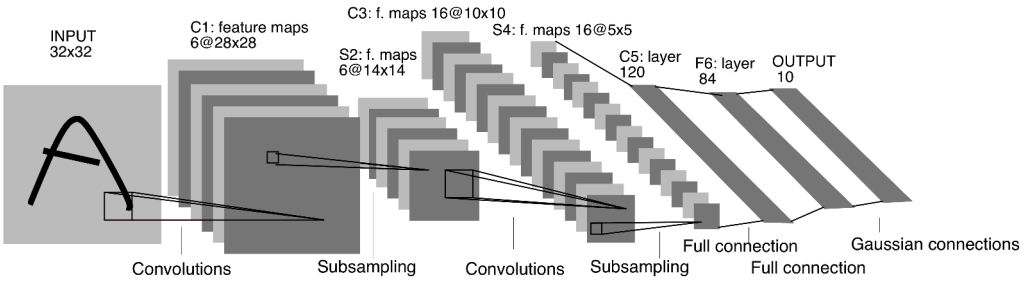


FIGURE 1.1: http://cs231n.github.io/convolutional-networks/

In this example, the input is an 32×32 image. Units in the first hidden layer are organized in 6 layers, each of which is a feature map. A unit in a feature map has 25 inputs connected to a 5×5 block in the input, called the **receptive field** of the unit. Each unit has 25 inputs, therefore 25 trainable weights plus a trainable bias.

The receptive fields of contiguous units in a feature map are centred on correspondingly contiguous units in the previous layer. Therefore overlapping occurs. It is important to note, that all units in the same feature map share the same weights, in particular the same 25 weights and bias. One reason for that is to reduce the number of parameters and another one, if some specific feature was already extracted at some place in the input, the same type of features could be detected at any place of the input using the same parameters. Say, if one found an edge at some place, one can then learn all other edges using the same set of weights. The other feature maps in the layer use different sets of weights and biases, thereby extracting different types of local features.

In the example of LeNet-5, at each input location six different types of features are extracted. A sequential scanning of the input image with a single unit that has a local receptive field and stores the states of the unit at corresponding locations in the feature map is equivalent to convolution, followed by additive bias and non-linear unit.

The kernel of convolution is a set of weights used by units in the feature map. An important property of convolutional layers is that if the input image is shifted, the output of the feature map is shifted by the same amount, but will be left unchanged. This is a basis of the resilience of CNNs to shifts and distortions.

1.2.1 Convolutional layer

Convolution is a mathematical operation on two functions (f and g) to produce a third function, that is typically viewed as a modified version of one of the original functions. Convolution is defined as follows:

$$(f \star g)(t) \stackrel{def}{=} \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau \tag{1.1}$$

In this context, the convolution formula can be decsribed as a weighted average of the function $f(\tau)$ ar the moment t where the weighting is given by $g(-\tau)$ simply shifted by amount t. As t changes, the weighting function emphasizes different parts of the input function. In the context of CNN, f is an input and g represents a kernel.

In two-dimensional image processing terms, the continuous convolution integral may be expressed as follows:

$$h(x,y) = (f \star g)(x,y) \stackrel{def}{=} \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(\tau_u,\tau_v)g(x-\tau_u, y-\tau_v)d\tau_u d\tau_v \tag{1.2}$$

In a manner analogous to one-dimesional convolution, the function $g(0-\tau_u, 0-\tau_v)$ is simply the image function $g(\tau_u,\tau_v)$ rotated by 180 degrees about the origin. The

function $g(x - \tau_u, y - \tau_v)$ is the function further translated to move the origin of the image function h to the point (x, y) in the (m, n) plane.

Convolution of digital sampled images is analogous to that for continuous images, except that the integral is transformed to a discrete summation over the image dimensions, m and n.

$$h(x, y) = (f \star g)(x, y) \stackrel{def}{=} \sum_{\tau_u} \sum_{\tau_v} f(\tau_u, \tau_v) g(x - \tau_u, y - \tau_v) \tag{1.3}$$

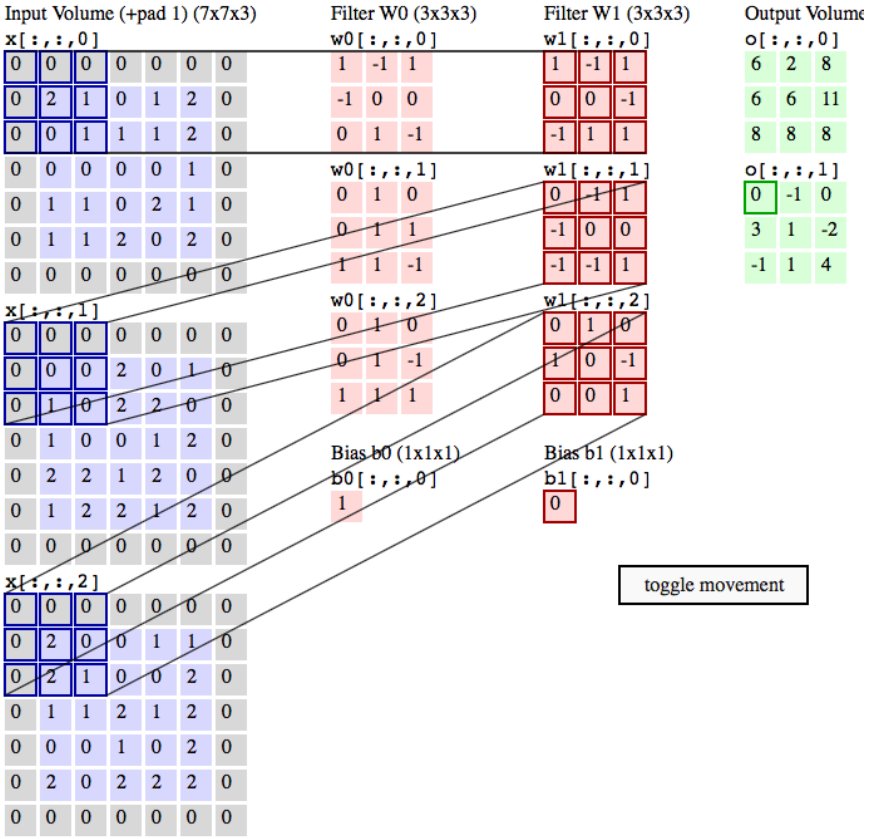


FIGURE 1.2: http://cs231n.github.io/convolutional-networks/

Here you can see an example (going through the link below the figure there is a nice demo), where input has depth 3 (three blue squares on the left) and there are 2 different filters (second and third column from the left, each of them has also depth 3) that are applied with a stride of 2 to each depthslice (each blue square in the first column individually) of the input. If we look at equation 1.3, then f is now three squares in the first column in figure 1.2 and g is at first the second and then the third column. Therefore, when applying the first filter, h is the first green square, in the last column of figure 1.2 and when applying the second filter to the input, the second green square

in the last column of figure 1.2.
Stride is a parameter which determines the value of the step while convolving the input with a filter. When the stride is 2, then the filters jump 2 pixels at a time as we slide them around.
Sometimes it will be convenient to pad the input volume with zeros around the border. The size of this **zero-padding** is a hyperparameter. The convenient feature of zero padding is that it will allow us to control the spatial size of the output volumes.
All together works as follows, starting with the first depthslice of the input (first square in the first column of figure 1.2) and the first depthslice of the first filter (first square in the second column of figure 1.2), we place the filter's slice over this first depthslice as shown in figure 1.2, and then moving the slice of the filter over the slice of the input from left to right and from top to bottom, for each second pixel (stride 2) performing element-wise multiplication and doing so for each slice of the input with the corresponding slice of the filter for each filter separately and then summing over the depth, we are getting, in this case, two output volumes, because there are two filters. This convolution operation results in output volume, which is represented in green. After this bias is added. This is how convolution layer functions in the example when the input is an image.

1.2.2 Pooling layer

Once a feature has been detected, its exact location becomes less important. Only its relative location to other features is relevant. Moreover, it is not only less important, but it can even be harmful, because the positions are likely to vary for different instances of the input. For example, if we take handwritten digits as input, and we will look at n different instances of the same digit, positions will vary drastically. A way out is a reduction of spatial resolution of the feature map. This can be achieved by means of a **pooling layer**, which performs local averaging and reduces sensitivity of the output to shifts and distortions. In LeNet-5 example, the second hidden layer is a pooling layer. This layer comprises six feature maps. The receptive field of each unit is a 2 by 2 area in the previous layer's corresponding feature map. Each unit computes the average of its four inputs, multiplies it by trainable weights, adds a trainable bias and passes the result through the non-linear unit, in particular a sigmoid function. A non-linear unit is needed in order to capture some non-linear patterns while training. Next to sigmoid function, **rectified linear unit (ReLU)** function is often used.

In the following figure, a max pooling operation is illustrated. In addition to max pooling, units can also perform other functions, such as average pooling or even L2-norm pooling. Average pooling was often used historically but has recently fallen out

of use in favor of the max pooling operation, which has been shown to work better in practice.

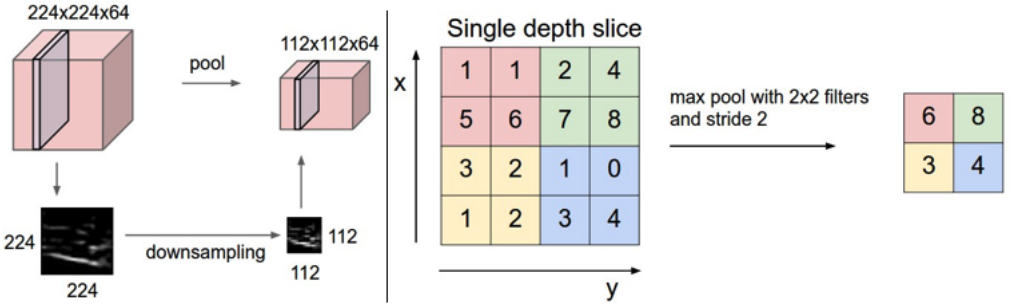


FIGURE 1.3: http://cs231n.github.io/convolutional-networks/

The pooling layer down-samples the volume spatially, independently in each depth slice of the input volume.

Here the difference between max- and average-pooling is illustrated:

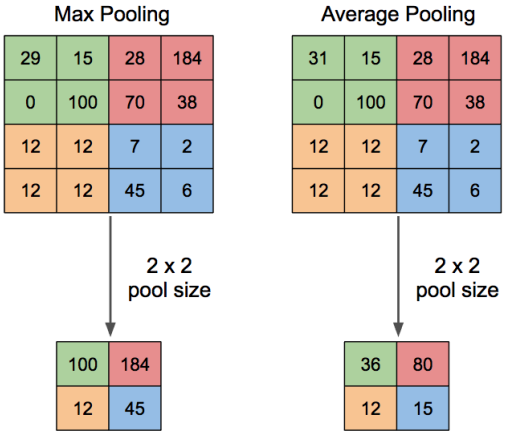


FIGURE 1.4: https://pythonmachinelearning.pro/introduction-to-convolutional-neural-networks-for-vision-tasks/

1.2.3 Fully connected (FC) layer

Neurons in a fully connected layer have full connections to all activations in the previous layer. The output from the convolutional layers represents high-level features in the data. While that output could be flattened and connected to the output layer, adding a fully connected layer is a cheap way of learning non-linear combinations of these features.

Convolutional and pooling layers are providing a meaningful, low-dimensional and in some sense invariant feature space and the fully connected layer is learning a (possibly non-linear) function in that space. It is also trivial to convert from FC layers to Convolutional layers.

1.2.4 Loss Function and its minimization

In order to train a neural network to perform some task, we must adjust the weights of each unit in a such way that the error between the desired output and the actual output is reduced.

Optimization algorithms used for training of deep models differ from traditional optimization algorithms. Machine learning usually acts indirectly. In most machine learning cases, we care about some performance measure P, that is defined with respect to the test set and may also be intractable. We therefore optimize P indirectly. We reduce a different cost function $J(\theta)$ in the hope that doing so will improve P. This is in contrast to pure optimization, where minimizing J is a goal in and of itself.

Without diving into rigorous details, for more insight one may want to look into [10], [11], we need some metric, in order to be able to analyze this error. For this reason, cost function is defined, it can be written as an average over the training set, such as

$$J(\theta) = \mathbb{E}_{(x,y)\sim\hat{p}_{data}} L(f(x,\theta), y), \tag{1.4}$$

where L is the per-example loss-function, $f(x,\theta)$ is the predicted output when the input is x, $\hat{p}_{data}$ is the empirical distribution, and θ are so called weights, or in other words parameters, which should be modified in order to find the solution. In the supervised learning case, y is a target output. It is trivial to extend this, for example, to include θ or x as arguments, or to exclude y as argument, to develop various forms of regularization or unsupervised learning.

Equation (1.4) defines an objective function with respect to the training set. We would usually prefer to minimize the corresponding objective function where the expectation is taken across the *data-generating distribution* p_{data} rather than just over the finite training set:

$$J^*(\theta) = \mathbb{E}_{(x,y)\sim p_{data}} L(f(x,\theta), y), \tag{1.5}$$

The goal of training a neural network is to reduce the expected generalization error given by equation (1.6). This quantity is known as **risk**. The expectation is taken over the true distribution p_{data}. If the true distribution would be known, risk minimization would be an optimization task solvable by an optimization algorithm. When we do not know $p_{data}(x,y)$ but only have a training set of samples, we have a machine learning problem.

In order to convert a machine learning problem back into an optimization problem is to minimize the expected loss on the training set. This means replacing the true

distribution $p(x, y)$ with the empirical distribution $\hat{p}(x, y)$ defined by the training set.

$$\mathbb{E}_{(x,y)\sim\hat{p}_{data}(x,y)}[L(f(x,\theta),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(x^i,\theta),y^i), \tag{1.6}$$

where m is the number of training examples.
The training process based on minimizing this average training error is known as **empirical risk minimization**.
Empirical risk minimization is prone to overfitting. Models with high capacity can simply memorize the training set. The most effective modern optimization algorithms are based on gradient descent, but many useful loss functions, such as $0-1$ loss, have no useful derivatives. These two problems force us to use a slightly different approach from empirical risk minimization, which in its turn brings another level of postponement, because now the quantity that we actually minimize is even more different from the quantity we truly want to minimize.
Insight into the optimization problem, and into the various techniques for solving it, can be obtained by considering a local quadratic approximation to the error function. It is possible to evaluate the gradient of an error function efficiently by means of the **backpropagation** procedure. The use of this gradient information can lead to significant improvements in the speed with which the minima of the error function can be located. The simplest way to use gradient information is to choose the weight update to comprise a small step in the direction of the negative gradient, so that

$$\theta^{\tau+1} = \theta^{\tau} - \eta\nabla E(\theta^{\tau}) \tag{1.7}$$

where $\eta > 0$ is known as the learning rate. After each such update, the gradient is re-evaluated for the new weight vector and the process is repeated. Note that the error function is defined with respect to a training set, and so each step requires that the entire training set be processed in order to evaluate ∇E. At each step the weight vector is moved in the direction of the greatest rate of decrease of the error function, and so this approach is known as **gradient descent**. Although such an approach might intuitively seem reasonable, in fact it turns out to be a poor algorithm, for reasons discussed in [12], there an in-depth discussion on which algorithms are more efficient can also be found.
Now we will take a look at an efficient technique for evaluating the gradient of an error function $E(\theta)$ for a feed-forward neural network. We will see that this can be achieved using a local message passing scheme in which information is sent alternately forwards and backwards through the network and is known as **error backpropagation**. Most training algorithms involve an iterative procedure for minimization of an error function, with adjustments to the weights being made in a sequence of steps. At each such step, we can distinguish between two distinct stages. In the first stage, the derivatives of

the error function with respect to the weights must be evaluated. In the second stage, the derivatives are used to compute the adjustments to be made to the weights. The simplest of such techniques involves gradient descent. It is important to recognize that the two stages are distinct. For in depth discussion on this matter, consult [11], which we are mainly following in this section.
The backpropagation procedure can be summarized as follows:

- Apply an input vector x_n to the network and forward propagate through the network using:
  - $a_j = \sum_i \theta_{ji} z_i$,
    where z_i is the activation, or input, that sends a connection to a unit j, and θ_{ji} is the weight associated with that connection.
  - $z_j = h(a_j)$

  to find the activations of the hidden and output units.
- Evaluate $\delta_k = \frac{\partial E_n}{\partial a_j}$ for all the output units using:
  - $\delta_k = y_k - t_k$,
    where y_k is the k-th output and t_k is the actual value.
- Backpropagate the δ's using:
  - $\delta_j = h'(a_j) \sum_k \theta_{kj} \delta_k$

  to obtain δ_j for each hidden unit in the network.
- Use $\frac{\partial E_n}{\partial \theta_{ji}} = \delta_j z_i$ to evaluate the required derivatives.

For batch methods, the derivative of the total error E can then be obtained by repeating the above steps for each pattern in the training set and then summing up over all patterns:

$$\frac{\partial E}{\partial \theta_{ji}} = \sum_n \frac{\partial E_n}{\partial \theta_{ji}} \tag{1.8}$$

Backpropagation is only the first stage and serves to evaluate the needed derivatives in order to use them in the second stage by feeding them into the chosen optimization algorithm, e.g. **Stochastic Gradient Descent**.

We will stop at this point; any further information on how cost function should be modified and which algorithms are then used for the minimization can be found in [10], [11].

1.2.5 Summary

When we saw an example of basic Convolutional Neural Network for digits classification task, a natural question is: how accurate are Deep Neural Networks on average? Since the emergence of Deep Neural Networks (DNNs) as a prominent technique in the field of computer vision, the ImageNet classification challenge has played a major role in advancing the state-of-the-art. While accuracy figures have steadily increased, the resource utilization of winning models has not been properly taken into account.
In the following [13] a comprehensive analysis of important metrics in practical applications is presented: accuracy, memory footprint, parameters, operations count, inference time and power consumption.
In the ImageNet classification challenge, the ultimate goal is to obtain the highest accuracy in a multiclass classification problem framework, regardless of the actual inference time.
In the following figure on the left, accuracy statistics for the major Deep Neural Network architectures is shown. The analysis is based on re-evaluations of top-1 accuracies for all networks with a single central-crop sampling technique. Single central-crop technique means that top-5 validation method is used for error measuring.
On the right, top-1 accuracy versus amount of operations required for a single forward pass is shown. For more statistical measurements see [13].

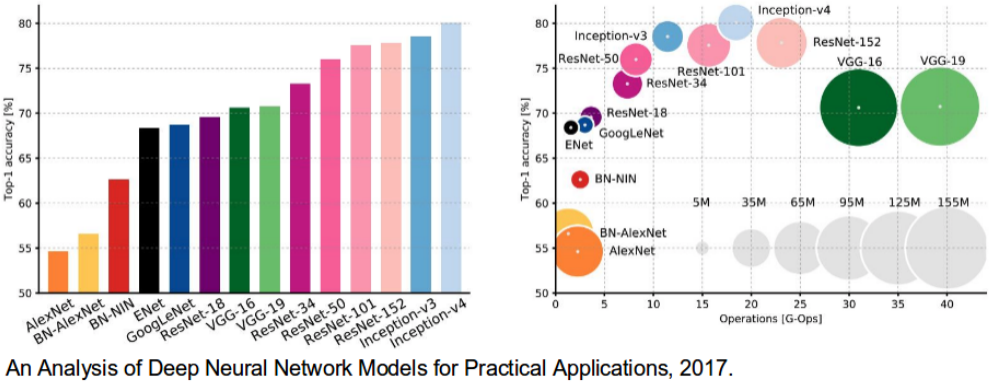


FIGURE 1.5: https://arxiv.org/pdf/1605.07678.pdf

The next question is, what are the other settings and data-types that CNNs could be applied to? One of the problem-types is learning from data that doesn't have a regular grid structure, with other words is non-Euclidean. In the next chapter we will look at this task in greater detail.

Chapter 2

Graph-convolutional networks

Neural networks that operate on graphs have previously been introduced in [14] and then in [15] as a form of recurrent neural network. Their framework requires the repeated application of contraction maps as propagation functions until node representations reach a stable fixed point. This approach went almost unnoticed, re-emerging in a modern form in [16]. The first formulation of CNNs on graphs is due to [17], who used the definition of convolutions in the spectral domain. For a more in-depth review of the history of CNNs on graphs, the reader is refered to [18].

As we already know, existing approaches to generalize convolutional networks to non-regular graph-structured data can be divided into two broad categories: spectral filtering methods and local filtering methods (also called spatial). While successful with noise-free data such as synthetic 3D shape models, spectral techniques have more difficulties with real observed data for which global decompositions may be unstable across, for instance, various shapes in various poses.

In the computer vision and graphics community, Masci et al. [8] showed the first CNN model on meshed surfaces, resorting to a spatial definition of the convolution operation based on local intrinsic patches. Followup works proposed a different construction of intrinsic patches on point clouds [7, 19]. Basically, all these methods differ in how they establish a correspondence between filter weights and nodes in local graph neighborhoods.

In contrast to the previously mentioned works, Masci's et al. method learns filter shapes by estimating the means and variances of Gaussians that associate filter weights to the local pseudo-coordinates. The approach that we are following [1] instead of considering hand-designed local pseudo-coordinates, uses the features of the preceding layer to map between local graph patches and filter weights.

2.1 Graph convolutions using dynamic filters

In order to see how this new approach corresponds to the conventional CNN, we will start with conventional architecture and will transform it step by step.

In the case of conventional CNNs, the parameters are represented as a set of $D \times E$ filters $F_{d,e}$, each of size $h \times w$, where D is the dimension of the input features, E the dimension of the output features. Therefore, in order to compute one of the E output channels, each of the D input channels should be convolved with corresponding filters, afterwards summed over D and biased.

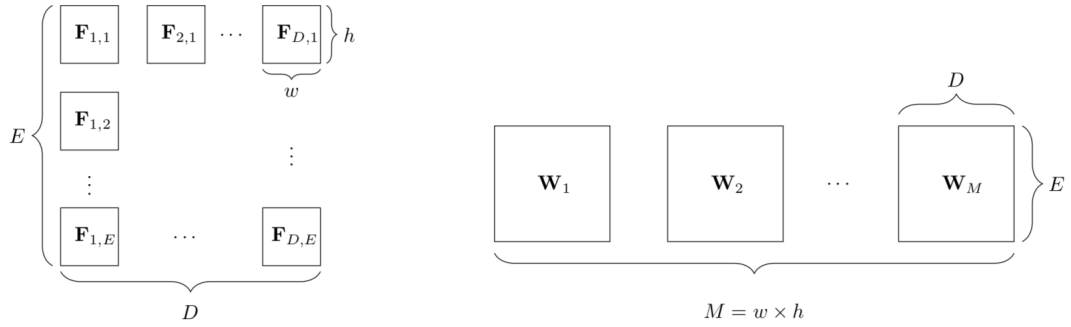


FIGURE 2.1: [1]. Left: representation as $D \times E$ filters $F_{d,e}$, each of size $h \times w$. Right: equivalent representation using $M = h \times w$ weight matrices, each of size $E \times D$.

An equivalent way to perform the same computation is to rearrange the weights as a set of $M = h \times w$ weight matrices $W_m \in \mathbb{R}^{E \times D}$. Each of these matrices projects input features in $\mathbb{R}^D$ to output features in $\mathbb{R}^E$. Projecting is here understood as vector matrix multiplication. The result of the convolution is derived by summing the projection of each of M neighbors of the central pixel. The output $y_i \in \mathbb{R}^E$ of the convolution can then be computed as follows:

$$y_i = b + \sum_{m=1}^{M} W_m x_{j(m,i)}, \tag{2.1}$$

where $b \in \mathbb{R}^{\mathbb{E}}$ denotes a vector of bias terms and $j(m,i)$ gives the index of the pixel in m-th relative position with respect to i. For example, $j(1,i) = i$ refers to the central pixel of the convolution, $j(2,i)$ refers to the top left pixel with respect to pixel i.

2.2 Generalization to non-Euclidean input domains

Current approach belongs to the local filtering approaches and the main difficulty in comparison to conventional CNNs is an establishment of correspondence between neighbors

and weight matrices $W_m \in \mathbb{R}^{\mathbb{E}\times\mathbb{D}}$. In regular CNNs this correspondence is straightforward and fixed, which is most important.

Before diving into an actual approach, introducing the settings is necessary. We assume that we have n samples $x_i \in \mathbb{R}^{d_x}$. Each sample x_i is associated with a vector $y_i \in \mathbb{R}^{d_y}$ for a regression task or a label $y_i \in \{0, , C\}$ for a classification task.

Input data is structured as a graph $G = (V, E, A)$, where V is the set of $|V| = d_x$ vertices, E is the set of edges and $A \in \mathbb{R}^{d_x \times d_x}$ is the adjacency matrix. This is called **signal classification / regression**, as the samples x_i to be classified or regressed are graph signals.

Other modelling settings are:

1. **node classification / regression**:
   Instead of considering signals on the graph, one can use a data graph, i.e. an adjacency matrix $A \in \mathbb{R}^{n\times n}$ which represents pairwise relationships between samples $x_i \in \mathbb{R}^{d_x}$. The problem here is to predict a graph signal $y \in \mathbb{R}^{n\times d_y}$ given a graph characterized by A and some graph signals $X \in \mathbb{R}^{n\times d_x}$.

2. **graph classification / regression**:
   Here interest is in classification of the whole graph, with or without signals on top. The task here is to classify or regress a whole graph $A_i \in \mathbb{R}^{n\times n}$ (with or without an associated data matrix $X_i \in \mathbb{R}^{n\times d_x}$) into $y_i \in \mathbb{R}^{d_y}$.

Settings for the current approach are a slight modification of signal classification/regression case. There are still n samples, but instead of $x_i \in \mathbb{R}^{d_x}$, we now have $x_i \in \mathbb{R}^{d_x \times D}$. In particular, this means that $|V|$ = number of features $= d_x$, so each vertex is a $D-dim$ feature, edges represent connections between features and on top of these graph we have n samples.

Now we start with the approach introduced in [1] itself. The main point is, instead of assigning each neighbor j of a vertex i to a single weight matrix as in the modified conventional CNN (2.1), soft-assignment $q_m(x_i, x_j)$ is used across the M weight matrices. Therefore, the function that maps the features from one layer to the next is defined as follows:

$$y_i = b + \sum_{m=1}^{M} \frac{1}{|\mathcal{N}_i|} \sum_{j\in\mathcal{N}_i} q_m(x_i, x_j) W_m x_j, \tag{2.2}$$

where $q_m(x_i, x_j)$ is the assignment of x_j to the m-th weight matrix, $\mathcal{N}_i$ is the set of neighbors of vertex i (including i) and $|\mathcal{N}_i|$ is it's cardinal.

As discussed above, there is no one-to-one mapping between the neighbor and the weight matrix. Instead, there is a soft-assignment function, which is some sort of distribution function and is defined as follows:

$$q_m(x_i, x_j) \propto exp(u_m^T x_i + v_m^T x_j + c_m) \tag{2.3}$$

with the following properties:

- $\sum_{m=1}^{M} q_m(x_i, xj) = 1$
- This formulation is resilient to variations in the degree of nodes, because:

$$\sum_{j \in \mathcal{N}_i} \frac{1}{|\mathcal{N}_i|} \sum_{m=1}^{M} q_m(x_i, x_j) = \sum_{j \in \mathcal{N}_i} \frac{1}{|\mathcal{N}_i|} = 1$$

- Nonetheless, conventional CNNs over grid-graphs are recovered if $\forall_i |\mathcal{N}_i| = M$ and the assignment is binary, i.e. $q_m(x_i, x_j) \in \{0, 1\}$.

It is also worth to note, that v_m and u_m are additional degrees of freedom, or in other words, weights that are also learned. In particular this feature distinguishes this approach from the other, because mapping between signals and weights is learned as an integral part of the network and is not deterministic.

In the following figure there is a schematic illustration of the mapping between neighbors x_j of the center pixel x_i and filter weights. On the left side in CNNs for pixel grids, on the right in graph-convolutional approach discussed above.

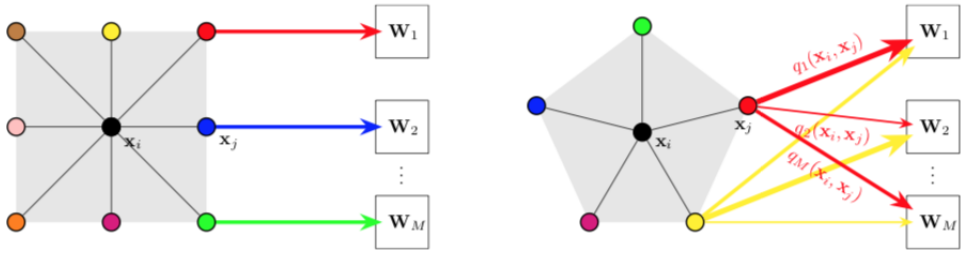


FIGURE 2.2: [1]

The key novelty in this approach is that the mapping between neighbors x_j of the center pixel x_i and filter weights is learned as an integral part of the neural network, using features computed in the preceding layer of the network, rather then using some fixed correspondence.

In the next figure difference in computations in a standard CNN and in the discussed graph approach is shown.

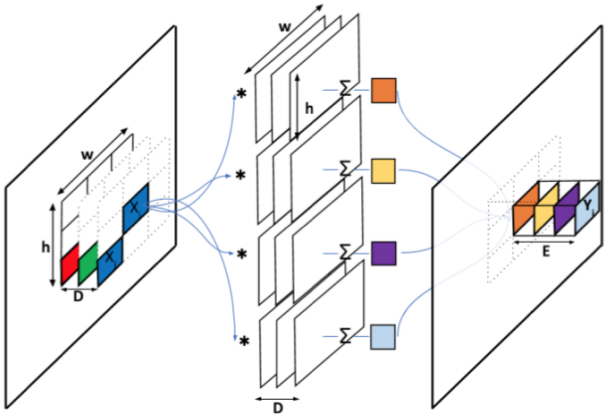


FIGURE 2.3: [1].Computation in a standard CNN where patches of $w \times h$ pixels are convolved with $D \times E$ filters to map the D dimensional input features to E dimensional output features.

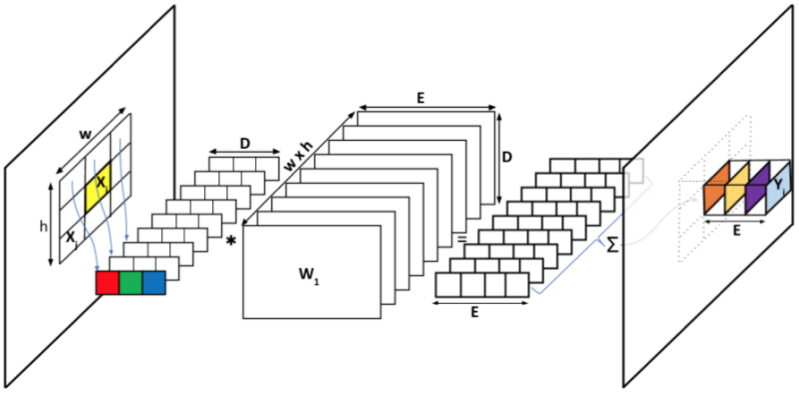


FIGURE 2.4: [1].Representing the CNN parameters as a set of $M = h \times w$ weight matrices, each of size $D \times E$. Each weight matrix is associated with a single relative position in the input batch.

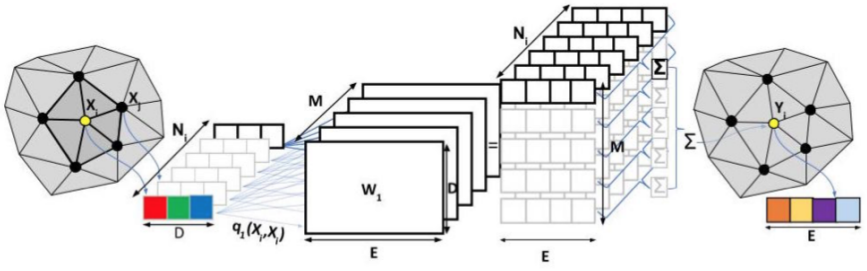


FIGURE 2.5: [1].Graph convolutional approach, where each relative position in the input is associated in a soft manner to each of the M weight matrices using the function $q(x_i, x_j)$.

Note, once more, that the main goal of the approach is to learn weight matrices W_m and additional weights v_m and u_m that come from soft-assignment functions q_m, in order to achieve high accuracy of prediction.

2.3 Computational complexity

Another important question is how the computational complexity of the new approach compares to the complexity of the standard CNN. The weight matrices W_m contain the same number of parameters in both, conventional and graph-based, CNN, in particular, MDE parameters, where M is number of weight matrices, D the input dimension and E the output dimension. Additional parameters in the graph-based approach with respect to standard CNN are vectors v_m and u_m, which contain $2MD$ parameters. Therefore, the total number of parameters increases by a factor $1+2/E$, because $(MDE+2MD = (MDE(1+2/E)))$.

According to equation 2.2, we need to multiply all feature vectors x_i with the weight matrices W_m and weight vectors v_m, u_m. This costs $\mathcal{O}(NMDE)$ operations, where N is the number of nodes in the graph. The soft-assignments in equation 2.3 and activations in 2.2 can be computed in $\mathcal{O}(NMKE)$ operations, where K is the average number of neighbors of each vertex. All together this results into $\mathcal{O}(NME(K+D))$.

For a conventional CNN, computational cost is $\mathcal{O}(NMED)$, see 2.1. Assuming that K is substantially smaller than D, which is typically the case, the computation cost of the graph-based approach is comparable to one of the standard CNN.

2.4 Translation invariance of soft-assignment in feature space

In equation 2.2, if $u_m = -v_m$, then $q_m^{ij} \propto exp(u_m^T(x_j - xi) + c_m)$, which leads to translation invariance of the q_m^{ij} in feature space. If the input features include spatial coordinates, it is natural to impose translation invariance on the assignment function. In the numerical experiments chapter, comparison of translation invariant and non-invariant cases will be shown.

Another point in equation 2.2 which can be tuned is, once more, soft-assignment function, but now instead of using just the linear sum and taking *exp*-function over this sum, one can use some other metric, for example soft-assignment based on Mahalanobis distance.

2.5 Outlook

In the next chapter, we will look at another reformulation of CNN for the graph-based data, this time, in the context of spectral graph theory. Afterwards, we will compare both approaches.

Chapter 3

Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering(ChebNet)

In contrast to the first approach, ChebNet is a spectral approach. This approach was introduced in [3] and from now on we will follow this paper. Because the approach is spectral, there is a well-defined localization operator on graphs via convolutions with Kronecker delta implemented in the spectral domain [20]. The convolution theorem [21] defines convolutions as linear operators that diagonalize in the Fourier basis (represented by the eigenvectors of the Laplacian operator).

3.1 Learning Fast Localized Filters

Settings for this approach are the same as for the previous one: an undirected, connected graph $G = (V, E, W)$, where V is a finite set of $|V| = n$ vertices, E is a set of edges and $W \in \mathbb{R}^{n \times n}$ is a weighted adjacency matrix encoding the connection weight between two vertices. In contrast to dynamic filters approach from previous chapter, adjacency matrix now plays an important role in the convolution operation.

3.1.1 Graph Fourier Transform

An input (signal) $x : V \to \mathbb{R}$ defined on the nodes of the graph may be seen as a vector $x \in \mathbb{R}^n$ where x_i is its value at the i^{th} node. A crucial operator in spectral graph analysis is the graph Laplacian [22], of which the combinatorical definition is $L = D - W \in \mathbb{R}^{n \times n}$

where $D \in \mathbb{R}^{n \times n}$ is the diagonal matrix with $D_{ii} = \sum_j W_{ij}$ and the normalized definition is $L = I_n - D^{-1/2} W D^{-1/2}$ where I_n is an identity matrix. L is a symmetric positive definite matrix, therefore it has set of orthonormal eigenvectors $\{u_l\}_{l=0}^{n-1} \in \mathbb{R}^n$ and their associated ordered real non-negative eigenvalues $\{\lambda_l\}_{l=0}^{n-1}$, identified as the frequencies of the graph. The Laplacian L is then diagonalized by the Fourier basis $U = [u_0, ..., u_{n-1}] \in \mathbb{R}^{n \times n}$ such that $L = U \Lambda U^T$ where $\Lambda = diag([\lambda_0, ..., \lambda_{n-1}]) \in \mathbb{R}^{n \times n}$.

The graph Fourier transform of a signal $x \in \mathbb{R}^n$ is then defined as $\hat{x} = U^T x \in \mathbb{R}^n$ and its inverse as $x = U\hat{x}$. Now we have all tools to define the filtering operation (convolution).

3.1.2 Spectral filtering of graph signals.

The convolution operator on graph $*_G$ is defined in the Fourier domain such that $x *_G y = U((U^T x) \odot (U^T y)$, where $\odot$ is the element-wise Hadamard product. Now, a signal x filtered by g_θ is defined as

$$y = g_\theta(L)x = g_\theta(U \Lambda U^T)x = U g_\theta(L) U^T x \tag{3.1}$$

A non-parametric filter, i.e all parameters are free, would be defined as

$$g_\theta(\Lambda) = diag(\theta), \tag{3.2}$$

where $\theta \in \mathbb{R}^n$ is a vector of Fourier coefficients.

3.1.3 Polynomial parametrization for localized filters.

There are two limitations to non-parametric filters:

- they are not localized in space
- their learning complexity is in $\mathcal{O}(n)$

These can be overcome with the use of a polynomial filter

$$g_\theta(L) = \sum_{k=0}^{K-1} \theta_k \Lambda^k, \tag{3.3}$$

where the parameter $\theta \in \mathbb{R}^K$ is a vector of polynomial coefficients. The value at vertex j of the filter g_θ centered at vertex i is given by $(g_\theta(L)\delta_i)_j = (g_\theta(L))_{i,j} = \sum_k \theta_k (L^k)_{i,j}$. By [[23], Lemma 5.2], $d_\mathcal{G}(i,j) > K$ implies $(L^K)_{i,j} = 0$, where $d_\mathcal{G}$ is the shortest path

distance, i.e. the minimum number of edges connecting two vertices on the graph. Therefore, spectral filters represented by K^{th}-order polynomials of the Laplacian are exactly K-localized and their learning complexity is $\mathcal{O}(K)$.

3.1.4 Recursive formulation for fast filtering.

The cost to filter a signal x as $y = Ug_\theta(\Lambda)U^T x$ is still high with $\mathcal{O}(n^2)$ operatons because of the multiplication with the Fourier basis U. A solution is to parametrize $g_\theta(L)$ as a polynomial function that can be computed recursively from L, as K multiplications by a sparse L cost $\mathcal{O}(K|E|) << \mathcal{O}(n^2)$. One such polynomial is the Chebyshev expansion.

A Chebyshev polynomial $T_k(x)$ of order k can be computed by recurrence relation $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$ with $T_0 = 1$ and $T_1 = x$. Chebyshev polynomials form an orthogonal basis for $L^2([-1,1], dy/\sqrt{1-y^2})$, the Hilbert space of square integrable functions respect to measure $dy/\sqrt{1-y^2}$. A filter can therefore be parametrized as the truncated expansion

$$g_\theta(\Lambda) = \sum_{k=0}^{K-1} \theta_k T_k(\widetilde{\Lambda}), \tag{3.4}$$

of order $K-1$, where the parameter $\theta \in \mathbb{R}^K$ is a vector of Chebyshev coefficients and $T_k(\widetilde{\Lambda}) \in \mathbb{R}^{n \times n}$ is the Chebyshev polynomial of order k evaluated at $\widetilde{\Lambda} = 2\Lambda/\lambda_{max} - I_n$, a diagonal matrix of scaled eigenvalues in $[-1, 1]$.

The filtering can then be computed as $y = g_\theta(L)x = \sum_{k=0}^{K-1} \theta_k T_k(\widetilde{\Lambda})x$. Setting $\bar{x}_k = T_k(\widetilde{\Lambda})x \in \mathbb{R}^n$, the recurrence relation can be used to compute $\bar{x}_k = 2\widetilde{\Lambda}\bar{x}_{k-1} - \bar{x}_{k-2}$ with $\bar{x}_0 = 1$ and $\bar{x}_1 = \widetilde{\Lambda}x$.

The entire filtering operation $y = g_\theta(L)x = [\bar{x}_0, ..., \bar{x}_{K-1}]\theta$ then costs $\mathcal{O}(K|E|)$.

The idea of recursive formulation is crucial when implementing ChebNet using TensorFlow. It allows to vectorize all the operations and as a consequence to leverage use of GPUs.

3.1.5 Tensorflow

Tensorflow is a machine learning software library released by Google. It allows the user to build and test different machine learning algorithms, some of which are already implemented and only user related changes are needed. In cases when some new changes are wanted, the user has access to all low level tools that are needed for this purpose.
A key feature of Tensorflow is that the main object one manipulates and passes around is the Tensor. A Tensor is a generalization of vectors and matrices to potentially higher

dimensions. Tensorflow represents tensors as n-dimensional arrays of base datatypes.
The use of tensors is inspired by the hardware that was also improved over the current decade and is usually used when working with machine learning algorithms, namely Graphical Processing Units or GPUs.
One can think of GPUs as "vector processors". Vector processors expect most of the computation to be expressed in terms of vector operations, and are optimized to perform these operations as quickly as possible, perhaps even at the expense of scalar performance.
Therefore, when one is developing machine learning algorithms, Tensorflow together with GPUs looks like a good setup to start with.

3.2 Graph Coarsening

Another question is how to perform pooling operations on graph structured data. In case of the standard CNN, pooling allows us to look at the data in different resolutions, so something similar is also needed here. It then requires some meaningful neighborhoods on graphs, where similar vertices are clustered together. Doing this multiple times is equivalent to multi-scale clustering of the graph. It is however known that graph clustering is NP-hard [24], so an approximation algorithm is needed. There are different approaches; one that was used in the paper we follow is called Graclu's multilevel clustering algorithm [25]. This algorithm uses a greedy scheme to compute coarser versions of a given graph and is able to minimize several popular clustering objectives, from which here normalized cut [26] is chosen. Graclu's greedy rule consists, at each coarsening level, of picking an unmarked vertex i and matching it with one of its unmarked neighbors j that maximizes the local normalized cut $W_{ij}(1/d_i + 1/d_j)$. The two matched vertices are then marked and the coarsened weights are set as the sum of their weights. This scheme divides the number of vertices at each level by approximately two (there exists a few singletons at each level, singletons are non matched vertices) from one level to the next.

After coarsening, the vertices of the input graph and its coarsened versions are not arranged in any meaningful way. It is possible to arrange the vertices in such a way that a graph pooling operation becomes as efficient as a 1D pooling. The first step is to create a balanced binary tree; the second step is to rearrange the vertices.

The next figure shows an example of **graph coarsening and pooling**:

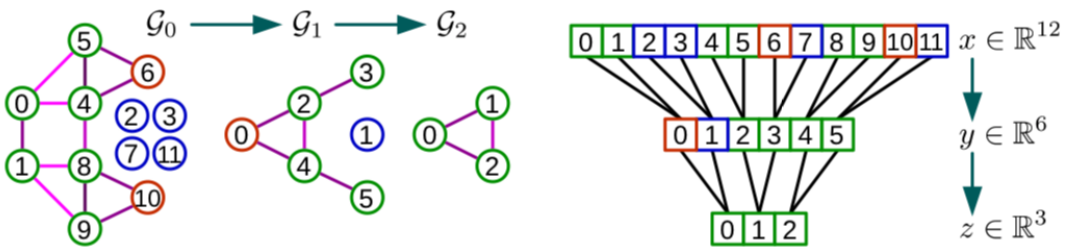


FIGURE 3.1: https://arxiv.org/pdf/1606.09375.pdf

In the above figure is an example of max pooling of size 4 (or two poolings of size 2) on a signal $x \in \mathbb{R}^8$ living on $\mathcal{G}_0$, the finest graph given as input. It originally has $n_0 = |V_0| = 8$ vertices, ordered arbitrarily. For a pooling of size 4, two coarsenings of size 2 are needed: Graclus gives $\mathcal{G}_1$ of size $n_1 = |V_1| = 5$, then $\mathcal{G}_2$ of size $n_2 = |V_2| = 3$, the coarsest graph. Sizes are set to $n_2 = 3$, $n_1 = 6$, $n_0 = 12$ and fake nodes (in blue) are added to V_1 (1 node) and V_0 (4 nodes) to pair with the singeltons (in orange), in such a way that each node has exactly two children. Nodes in V_2 are then ordered arbitrarily and nodes in V_1 and V_0 are ordered consequently. At that point, the arrangement of vertices in V_0 permits a regular 1D pooling on $x \in \mathbb{R}^{12}$ such that $z = [max(x_0, x_1), max(x_4, x_5, x_6), max(x_8, x_9, x_{10})] \in \mathbb{R}^3$, where the signal components x_2, x_3, x_7, x_{11} are set to neutral value.

Chapter 4

Numerical Experiments

In this chapter we will look into some results that were obtained while training presented architectures and we will compare these results to those presented in the corresponding papers. We will also discuss some further steps which might lead to better performance in speed and accuracy.

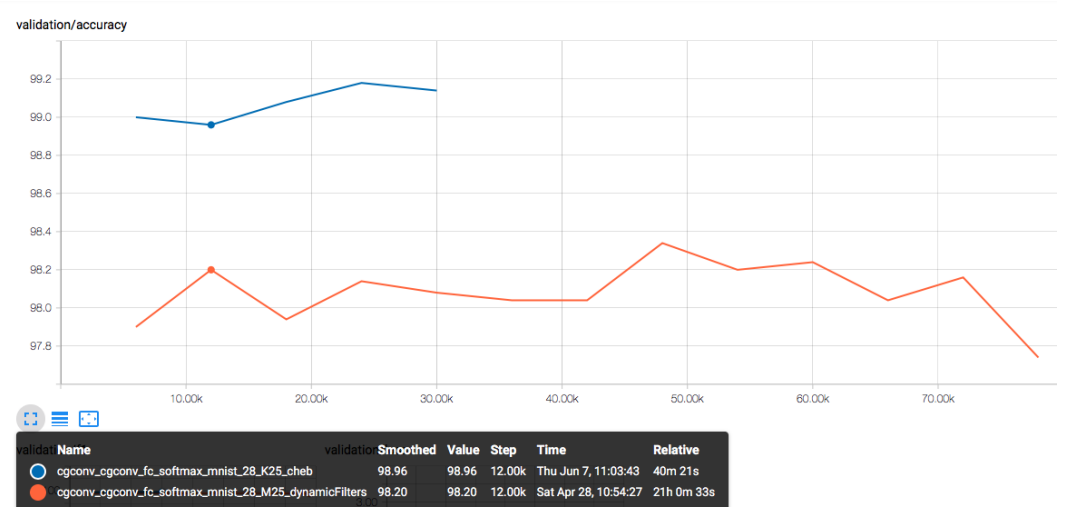
4.1 MNIST Dataset

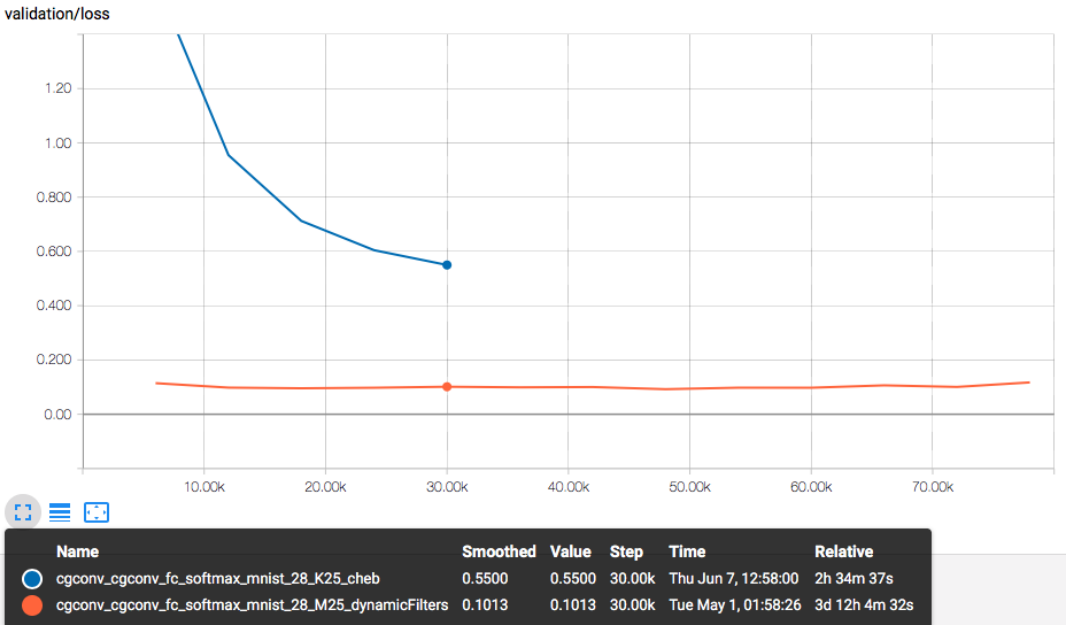
To validate the approach presented in Section 2.1, we used a dataset of 70,000 digits represented on a 2D grid of size 28×28. First we had to represent each image as a graph. For this reason we construct an 8-NN graph of the 2D grid which produces a graph of $n = |V| = 976$ nodes ($28^2 = 784$ pixels and 196 fake nodes as explained in Section 3.2) and $|E| = 3198$ edges.

In the next figure, there is a plot of validation accuracy for both approaches presented above. In case of ChebNet $K = 25$, it means filters are of the size 5×5, and for DynamicFilters approach $M = 25$, which corresponds to the number of weight matrices used per layer. For the relation between M and K, refer to Section 2.2. In the table below, the figure there is all-important information for both approaches. Another important point is the time it takes to achieve shown accuracies. In both cases Adam [27] optimization algorithm together with mini-batch approach was used. Training was performed on a CPU-cluster.

Compute Node	Dell PowerEdge M620 et	(Total: 78)
CPUS	Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz	(Total: 256)
Memory	64 GB per node	(Total: 4992 GB)
Hard disk	500 GB per node	

Such a big time difference (actual time below the first graph) lies in implementation details, namely in using fast filtering technique presented in Section 3.1.4 for ChebNet and in case of DynamicFilters there is no similar technique, besides vectorization, which is needed when using GPUs, but when implementing vectorization, it brings redundancy (the same parameters should be stored and calculated multiple times in order to achieve vectorization), which is one of the trade-offs between using of GPUs (potentially faster calculations) and increased number of parameters (increasing storage capacity) that should be considered.



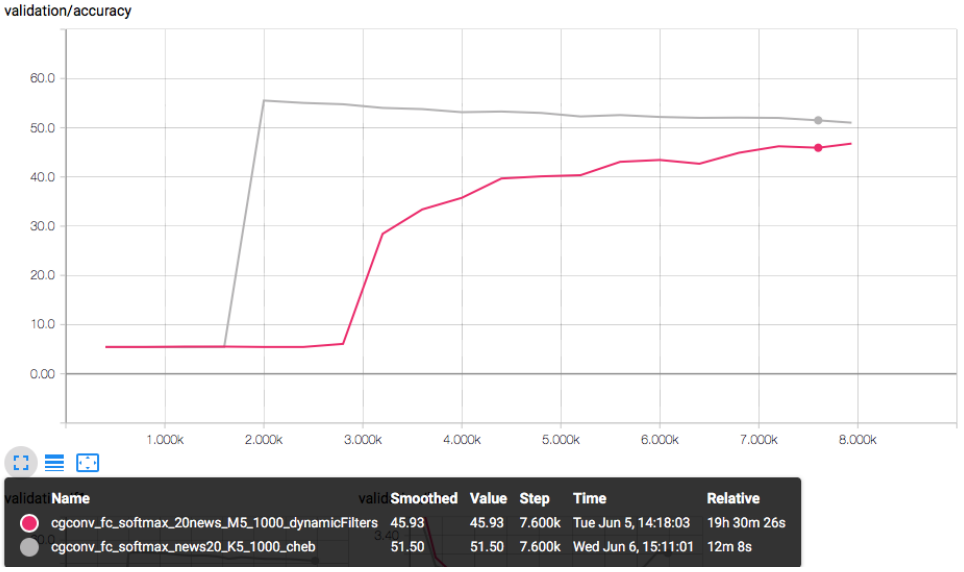


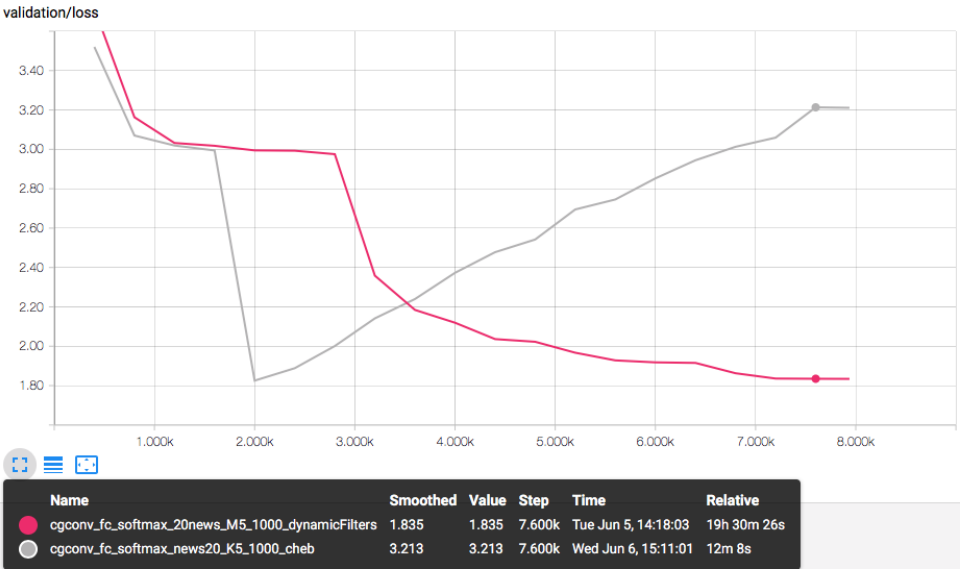
In the following table, all-important details of MNIST test are presented.

Dataset	Architecture	Approach	Accuracy	Computation cost	Number of parameters
MNIST	GC32-P4-GC64-P4-FC512	Chebyshev (K=25)	99.1	O(NKED)	KDE
MNIST	GC32-P4-GC64-P4-FC512	Dynamic Filters (M=25)	98.2	O(NME(avg.Number.Neighb.+D))	MDE + 2MD

4.2 Text Categorization on 20NEWS Dataset

In the previous section, the graph was generated from structured data; in this example graph is generated from unstructured data. Now we are dealing with a text categorization problem in the 20NEWS dataset which consists of 18,846 (11,314 for training and 7,532 for testing) text documents associated with 20 classes. We extracted the 1,000 most common words from the 93,953 unique words in this corpus. Each document is represented using the bag-of-words model, normalized across words. To be able to test both models, we construct a 16-NN graph in the same manner as with MNIST dataset, with z_i being the word2vec embedding, where z_i is the 2D coordinate of word i, which results in a graph of $n = |V| = 1,000$ nodes and $|E| = 14,346$ edges. Models were trained by the Adam optimizer with an initial learning rate of 0.001.





In the following table, all-important details of 20NEWS example are presented.

Dataset	Architecture	Approach	Accuracy
20NEWS	GC32	Chebyshev (K=5)	53.1
20NEWS	GC32	Dynamic Filters (M=5)	46.2

4.3 Conclusion and future work

In this work, we have looked into recent papers about generalization of CNNs to graphs. The first approach, which uses dynamic filters, belongs to spatial approaches, the second one, ChebNet, is a spectral approach. In the following table we summarize both approaches:

- ChebNet
  - Filters are exactly localized in r-hops support
  - $\mathbb{O}(1)$ parameters per layer
  - No computation of $\phi, \phi^T \Rightarrow \mathbb{O}(n)$ computational complexity (assuming sparsely-connected graphs)
  - Because of the previous statement, efficient vectorization while implementing in Tensorflow
  - Stable under coefficients perturbation
  - Filters are basis-dependent $\Rightarrow$ do not generalize across graphs

- Dynamic filters
  - Local filters are determined dynamically, based on the features in the preceding layer of the network
  - Stable under coefficients perturbation
  - For vectorization while implementing redundancy is needed

However, it is difficult to come to any conclusions at this point about which approach is more accurate and computationally more efficient. For this, further experiments using GPUs are needed.

4.3.1 Tensortrain

As we saw in 4.1, the training time was longer in case of Dynamic Filters approach. Therefore, it might be an advantage to use Tensor Train decomposition which was presented in [28]. Below are the results in ms benchmarking T3F on CPU and GPU and comparing against the TTPY library.

Op	TTPY 1 object CPU	T3F 1 object CPU	T3F 1 object GPU	T3F 100 objects CPU	T3F 100 objects GPU
matvec	11.142	0.129	0.121	0.003	0.003
matmul	86.191	0.125	0.133	0.004	0.004
norm	3.790	1.902	0.893	0.422	0.050
round	73.027	0.159	0.165	0.006	0.006
gram	0.145	0.806	0.703	0.029	0.001
project	116.868	1.564	1.658	0.017	0.018

FIGURE 4.1: https://github.com/Bihaqo/t3f

The main idea behind Tensor Train decomposition is to present each tensor in a specific form and then perform all kinds of operations and because of this, specific representation operations will take less time.

References

[1] Nitika Verma, Edmond Boyer, and Jakob Verbeek. Feature-steered graph convolutions for 3d shape analysis. 2018. URL https://arxiv.org/pdf/1706.05206.pdf.

[2] Yann Lecun, Lon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998. URL http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

[3] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. 2016. URL https://arxiv.org/abs/1606.09375.

[4] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann Lecun. Spectral networks and locally connected networks on graphs. In *International Conference on Learning Representations (ICLR2014), CBLS, April 2014*, 2014. URL https://arxiv.org/pdf/1312.6203.pdf.

[5] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *CoRR*, abs/1506.05163, 2015. URL https://arxiv.org/pdf/1506.05163.pdf.

[6] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016. URL https://arxiv.org/pdf/1609.02907.pdf.

[7] Davide Boscaini, Jonathan Masci, Emanuele Rodolà, and Michael Bronstein. Learning shape correspondence with anisotropic convolutional neural networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 3189–3197. Curran Associates, Inc., 2016.

[8] Jonathan Masci, Davide Boscaini, Michael M. Bronstein, and Pierre Vandergheynst. Geodesic convolutional neural networks on riemannian manifolds. In *Proceedings of the 2015 IEEE International Conference on Computer Vision Workshop (ICCVW)*,

ICCVW '15, pages 832–840, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-9711-7. doi: 10.1109/ICCVW.2015.112. URL http://dx.doi.org/10.1109/ICCVW.2015.112.

[9] Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodolà, Jan Svoboda, and Michael M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *CVPR*, pages 5425–5434. IEEE Computer Society, 2017. URL https://arxiv.org/pdf/1611.08402.pdf.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[11] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

[12] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA, 1995. ISBN 0198538642.

[13] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *CoRR*, abs/1605.07678, 2016. URL http://arxiv.org/abs/1605.07678.

[14] Michele Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, 2:729–734 vol. 2, 2005. URL https://www.researchgate.net/profile/Franco_Scarselli/publication/4202380_A_new_model_for_earning_in_raph_domains/links/0c9605188cd580504f000000/A-new-model-for-earning-in-raph-domains.pdf.

[15] Franco Scarselli, Michele Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *2009 IEEE Transactions on Neural Networks, 2009.*, 20(1):61–80, 2009. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1015.7227&rep=rep1&type=pdf.

[16] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. *CoRR*, abs/1511.05493, 2015. URL http://dblp.uni-trier.de/db/journals/corr/corr1511.html#LiTBZ15.

[17] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *CoRR*, abs/1312.6203, 2013.

[18] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Process. Mag.*, 34(4):18–42, 2017.

[19] Davide Boscaini, Jonathan Masci, Emanuele Rodol, Michael M. Bronstein, and Daniel Cremers. Anisotropic Diffusion Descriptors. *Computer Graphics Forum*, 2016. ISSN 1467-8659. doi: 10.1111/cgf.12844.

[20] David I. Shuman, Sunil K. Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE Signal Process. Mag.*, 30(3):83–98, 2013. URL http://dblp.uni-trier.de/db/journals/spm/spm30.html#ShumanNFOV13.

[21] Stphane Mallat. *A Wavelet Tour of Signal Processing, Third Edition: The Sparse Way*. Academic Press, Inc., Orlando, FL, USA, 3rd edition, 2008. ISBN 0123743702, 9780123743701.

[22] F. R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.

[23] David K. Hammond, Pierre Vandergheynst, and Rémi Gribonval. Wavelets on graphs via spectral graph theory. *CoRR*, abs/0912.3848, 2009.

[24] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Inf. Process. Lett.*, 42(3):153–159, May 1992. ISSN 0020-0190. doi: 10.1016/0020-0190(92)90140-Q. URL http://dx.doi.org/10.1016/0020-0190(92)90140-Q.

[25] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29 (11):1944–1957, November 2007. ISSN 0162-8828. doi: 10.1109/TPAMI.2007.1115. URL http://dx.doi.org/10.1109/TPAMI.2007.1115.

[26] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):888–905, August 2000. ISSN 0162-8828. doi: 10.1109/34.868688. URL https://doi.org/10.1109/34.868688.

[27] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://dblp.uni-trier.de/db/journals/corr/corr1412.html#KingmaB14.

[28] Alexander Novikov, Pavel Izmailov, Valentin Khrulkov, Michael Figurnov, and Ivan Oseledets. Tensor train decomposition on tensorflow (t3f). *arXiv preprint arXiv:1801.01928*, 2018.