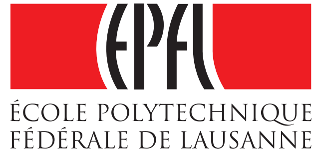


Studying graph convolutional neural networks

Thomas Grivaz

School of Computer and Communication Sciences

Semester Project

December 2016

Responsible
Prof. Pierre Vandergheynst
EPFL / LTS2

Supervisors
Michaël Defferrard
Nathanaël Perraudin
EPFL / LTS2

Contents

1 Introduction **1**

2 Background **1**

2.1 Signal Processing on Graphs 1

2.2 Stationary signal on Graphs 2

2.3 Graph convolutional neural networks 2

3 Framework description **4**

3.1 tools 5

3.2 Synthetic dataset 5

4 Experiments **7**

4.1 Assertion of the ground truth 7

4.2 Fixing the Chebyshev coefficients of the filters 8

4.3 Fixing the weights of the fully connected layer 9

4.4 Tests with different initializations and regularizations 9

5 Conclusion **12**

1 Introduction

Convolutional neural networks have been used for a number of years in various applications. They are particularly efficient to detect and extract features that lie on a regular euclidian domain such as images or sound. But not all data are confined to such regular grids, in fact many important real-world datasets are now in the form of graphs or networks such as social networks, protein-interaction networks, communication or transportation networks and so on. Yet it's not until recently that work has been devoted to adapt convolutional neural networks on such datasets. As this is a recent field, there is no established literature on how graph convolutional neural networks work and what make them work. In this report, we try to answer those questions and bring an intuitive and in depth explanation of the behaviour of a graph convolutional neural network with help of a synthetic dataset.

What makes standard convolutional neural networks good in the first place is their ability to detect local patterns in the data and combine them to provide high-level structures. At the core of this success is the stationary property of the input: filters learn local features shared across the input regardless of their exact spatial location. In this case stationarity is well defined via the translation operator on the grid. The first challenge here is how to generalize this property to graphs and exploit it.

The second important characteristic of CNNs is the convolution operator. In the 2D domain a localized kernel is slid across the input performing point-wise multiplications resulting in feature maps. Bringing convolution to graphs is not straight-forward as we have to take into account the properties of the operation: how can we define localized filters? How can we spatially cover the input? How do we define graph convolution? We'll answer those questions thanks to prior works in the domain. Less formally, the role of filters in standard CNNs is figured out. If we take the example of images, generally the first convolutional layer acts as an edge detector. We will also explain what represent the learned filters in a graph convolutional neural network.

Finally we will try to characterize the problem to be optimized i.e. what the optimal solution would be, how we can help the network converging to this solution by trying different initializations and regularizations.

2 Background

In this section we review the theory upon which graph convolutional neural networks are based.

2.1 Signal Processing on Graphs

Signal processing on graphs is a novel field that merges algebraic and spectral graph theoretic concepts with computational harmonic analysis to process such signals on graphs. A more in depth analysis of the field can be found in [1], we will state here the main concepts and formulas.

Weighted graphs and graph signals
The data we analyzed during this project were undirected, connected, weighted graphs. Such graphs are defined as $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, \mathbf{W}\}$ where $\mathcal{V}$ is the set of vertices with $|\mathcal{V}| = N$, $\mathcal{E}$ is the set of edges and $\mathbf{W}$ is the weighted adjacency matrix. A signal or function $f : \mathcal{V} \to \mathbb{R}$ defined on the vertices of the graph may be represented as a vector $\mathbf{x} \in \mathbb{R}^N$ where x_i represents the signal value at the ith vertex.

Graph Laplacian and Graph Fourier transform

The combinatorial graph Laplacian is defined as $\mathbf{L} = \mathbf{D} - \mathbf{W} \in \mathbb{R}^{N \times N}$. Where $\mathbf{D}$ is the diagonal degree matrix. The graph Laplacian is a real symmetric matrix, thus it has a complete set of orthonormal eigenvectors denoted by $\{\mathbf{u}_l\}_{l=0}^{n-1} \in \mathbb{R}^N$ with associated real, non-negative eigenvalues $\{\lambda_l\}_{l=0}^{n-1}$. The normalized Laplacian is defined as $\mathbf{L} = \mathbf{I} - \mathbf{D}^{-1/2}\,\mathbf{W}\,\mathbf{D}^{-1/2}$, satisfying $0 \leq \lambda_l \leq 2$. Using the eigendecomposition of a matrix, we can write $\mathbf{L} = \mathbf{U}\Lambda\mathbf{U}^T$ where $\mathbf{U}$ is the Fourier basis and Λ is the diagonal matrix with ordered eigenvalues on its diagonal.

In classical Fourier analysis, complex exponentials associated with low frequencies are smooth and slowly oscillating while high frequency complex exponentials oscillate rapidly. In graph theory, we have an equivalent notion by means of the eigenvalues and eigenvectors. Eigenvectors associated to low eigenvalues λ_l behave smoothly across the graph while eigenvectors associated to high eigenvalues oscillate more rapidly.

The graph Fourier transform of a signal $\mathbf{x}$ is defined as $\hat{\mathbf{x}} = \mathbf{U}^T\mathbf{x}$. Its inverse graph Fourier transform is then given by $\mathbf{x} = \mathbf{U}\hat{\mathbf{x}}$. This operation enables us to represent a signal either in the vertex domain or in the graph spectral domain.

Convolution and graph spectral filtering

Because of the irregular structure of a graph, there is no notion of translation of a signal across the vertex set but we can define the convolution between two graph signals as a point-wise multiplication in the Fourier domain, similarly to a classic time signal. For two graph signals $\mathbf{x}, \mathbf{y}$, it follows that :

$$\mathbf{x} *_{\mathcal{G}} \mathbf{y} = \mathbf{U}((\mathbf{U}^T\mathbf{x}) \odot (\mathbf{U}^T\mathbf{y}))$$

We define graph spectral filtering as:

$$\mathbf{y} = h(\mathbf{L})\mathbf{x} = h(\mathbf{U}\Lambda\mathbf{U}^T)\mathbf{x} = \mathbf{U}h(\Lambda)\mathbf{U}^T\mathbf{x}$$

where $h(.)$ is the transfer function of the filter and $h(\Lambda)$ is a diagonal matrix with entries $h(\lambda_l)$.

2.2 Stationary signal on Graphs

The stationary property of data is a very important property that will become handy later on. From [6], a stochastic graph signal $\mathbf{x}$ defined on the vertices of a graph $\mathcal{G}$ is called Graph Wide-Sense Stationary (GWSS). if and only if it satisfies the following properties:

1. its first moment is constant over the vertex set: $m_{\mathbf{x}}[i] = \mathbb{E}\{\mathbf{x}[i]\} = c \in \mathbb{R}$.
2. its covariance matrix $\Sigma_{\mathbf{x}}[i,j]$ is jointly diagonalizable with the Laplacian of $\mathcal{G}$, i.e. $\Sigma_{\mathbf{x}} = \mathbf{U}\mathbf{\Gamma}_{\mathbf{x}}\mathbf{U}^T$, where $\mathbf{\Gamma}_{\mathbf{x}}$ is a diagonal matrix.

This definition implies that the spectral components of $\mathbf{x}$ are uncorrelated. We also state here an important theorem: *When a graph filter h is applied to a GWSS signal, the result remains GWSS.*

2.3 Graph convolutional neural networks

Graph convolutional neural networks (GCNNS) extend the notion of CNNS by learning filters on irregular domains. All theoretical concepts are based on [2]. We present here layer by layer how they operate.

Data

The network takes as input a data matrix $\mathbf{X} \in \mathbb{R}^{n \times N}$ where rows are individual observations and columns are features. This matrix bears information content related to the underlying graph (e.g. signals having different frequency contents). Note that we have as many features as nodes in the graph. Besides this matrix, we will also use a representative description of the graph in matrix form, typically the adjacency matrix $\mathbf{A}$ or the Laplacian $\mathbf{L}$, both of size $N \times N$.

For a signal classification or regression task, our goal is to predict a discrete or continuous label y_i for each sample, respectively. This is the assignment we will focus on in this project. Other tasks include node classification/regression where we have to predict labels for nodes or graph classification/regression: given a collection of graphs, the goal is to learn a function that can be

used on unseen graphs. More informations on those problems can be found in [3]-[4].

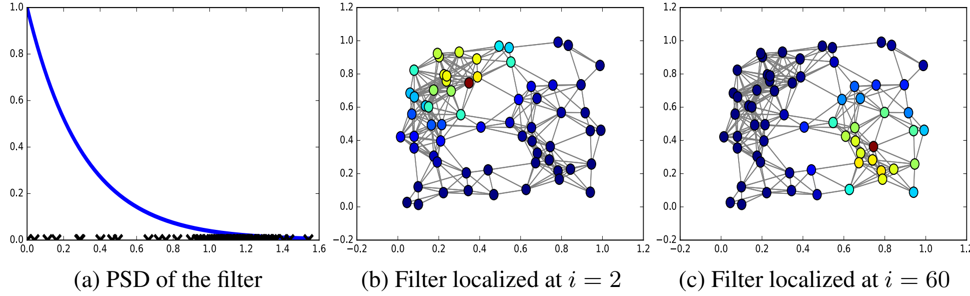
Convolutional layer

Before we get into details on the convolutional layer. Let us first state that there are two ways to convolve the input with a filter: either in spatial domain or in frequency/spectral domain. We only used the later approach in this project. We would like our filters to have two desirable properties: locality and low computational complexity. We previously defined the (non-parametric) filtering operation as $\mathbf{y} = \mathbf{U}h(\Lambda)\mathbf{U}^T\mathbf{x}$. The issues with a non-parametric filter are that they are not localized in space and learning takes $\mathcal{O}(N)$ steps, the dimensionality of the graph.

The locality problem can be overcome by using Laplacian-based polynomial spectral filters instead, defined as:

$$h_\theta(\Lambda) = \sum_{k=0}^{K-1} \theta_k \Lambda^k$$

With $\theta \in \mathbb{R}^K$ being a vector of polynomial coefficients. An interesting property of the spectral filtering approach is that we can naturally define the localization by a Kronecker delta function. Given a kernel h_θ, we can localize this kernel at vertex i with the operation $h_\theta(\mathbf{L})\delta_i$ i.e. the value at vertex j of the filter h_θ centered at vertex i is given by $h_\theta(\mathbf{L})[i, j]$. An example is displayed on figure 1.



(a) PSD of the filter (b) Filter localized at $i = 2$ (c) Filter localized at $i = 60$

Figure 1: Example of different localizations of a filter

figure 1a shows the PSD of the filter defined on a sensor graph of 64 nodes, here a heat kernel defined as $h(x) = e^{(-\tau x)}$ with $\tau = 5$ was used. The heat kernel represents the evolution of temperature in a region whose boundary is held fixed at a particular temperature (typically zero), such that an initial unit of heat energy is placed at a point at time $t = 0$, which will be the center of the filter in our case. Figures 1b and 1c show the filter plotted on the graph and localized at different vertices.

Moreover these filters are localized in a ball of radius K, provided that the filter is a Kth-order polynomial of the Laplacian. An extensive proof of this result can be found in [5]. Learning these filters boils down to learning coefficients θ_k, thus the learning complexity if $\mathcal{O}(K)$ (same as standard CNNS).

Now that we have overcome the first problem, the locality, by using parametric spectral filters, there is still one issue to tackle: the computational complexity. Indeed the filtering operation still implies to compute the eigenvalue decomposition of the Laplacian, but also two successive matrix multiplications (graph Fourier transform and inverse one) which each take $\mathcal{O}(n^2)$ steps. A solution of this problem brought in [2] is to recursively compute the polynomial function by a Chebyshev expansion of the Laplacian. Such operation has a cost of $\mathcal{O}(K|\mathcal{E}|) \ll \mathcal{O}(n^2)$. The resulting filter is as follows:

$$h_\theta(\mathbf{\Lambda}) = \sum_{k=0}^{K-1} \theta_k T_k(\tilde{\mathbf{\Lambda}})$$

T_k are Chebyshev polynomials of orker k, computed recursively with the formula $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$ with $T_0 = 1$ and $T_1 = x$, $\tilde{\mathbf{\Lambda}}$ is the diagonal matrix of scaled eigenvalues i.e. $\tilde{\mathbf{\Lambda}} = 2\mathbf{\Lambda}/\lambda_{max} - \mathbf{I}_n$ so that the eigenvalues are within the $[-1, 1]$ range.

Learning filters and producing feature maps, in short
Now that we saw the technique used to compute these filters, here's what happens in the convolutional layer:

- For layer j, a batch input volume $\mathbf{X}^{(j)}$ of dimensions $n \times p \times F^{(j-1)}$ is fed to the filters, n being the number of observations in this batch, p the number of nodes of the graph and $F^{(j-1)}$ the depth of the previous layer.
- for each filter i:
  - each depth slice $\mathbf{X}_s^{(j)} \in \mathbb{R}^{n \times p}$ of $\mathbf{X}^{(j)}$ is fetched to the filter i, resulting in the output $\mathbf{y}_i = h_{\theta_i}(\mathbf{L})(\mathbf{X}_s^{(j)})^T = \sum_{k=0}^{K-1} \theta_{i,k} T_k(\tilde{\mathbf{L}})(\mathbf{X}_s^{(j)})^T$.
  - the learned parameters are the Chebyshev coeffcients $\theta_{i,k}$. For $F^{(j)}$ filters at layer j, each of polynomial order K, this amounts to $F^{(j-1)} \times F^{(j)} \times K$ weight parameters to learn.
  - Note that in practice, the splitting is actually not done this way, the input volume is reshaped differently to take advantage of tensor operations and compute the filtering only once.
- After filtering, the feature maps are passed through a non-linearity, typically a ReLU.
- Graph is coarsened and pooled (not covered in this project, we advise the reader to refer to [2]).

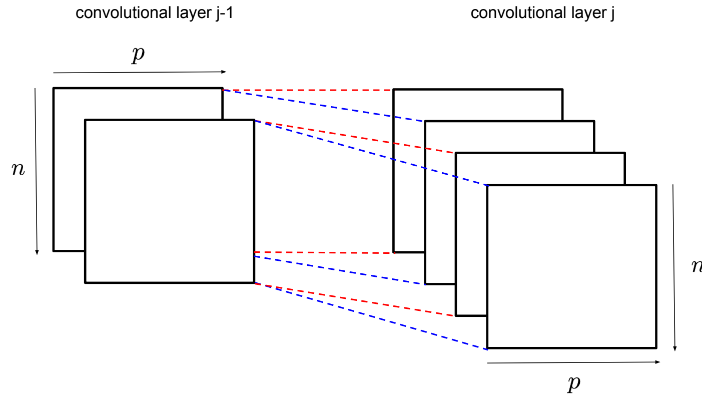


Figure 2: Visual representation of the convolutional layer. The output of layer $j-1$ is a volume $n{\times}p{\times}2$. Assume that we have two filters for layer j, each dashed line represents a filtering operation, red being filter 1 and blue filter 2. Thus we have $K \times 2 \times 2$ Chebyshev coefficients and the output volume will be $n \times p \times (2 \times 2)$

Fully connected layer
The fully connected layer is the same as in any standard CNN: every neuron in the previous layer is connected to every neuron of the next layer and followed by an activation function. The role of the fully connected layer is to create high-level features by learning non-linear combinations of the features learned at the previous layer(s).

3 Framework description

In this section we briefly describe the tools and libraries we used to conduct our analysis.

3.1 tools

PyGSP
PyGSP[1] is a graph processing toolbox implemented in Python. This toolbox provides many features such as easy graph creation from a list of pre-constructed graphs, signal processing tasks like filtering and so on.

cnn_graph
This repository[2] is the implementation of a functional graph convolutional neural network model.

3.2 Synthetic dataset

Since our goal here is to study the inner working and performance of a graph convolutional neural network, we needed a dataset that exhibited certain properties and complied to constraints:

- The most important feature is that the data has to be stationary i.e. the output is not dependent of the spatial location on the signal on the graph. Similarly to an object recognition task where the label is independent of the position of the object(s) in the image, we need stationarity to be able to study the data independently of its localization.
- Given our small processing power, training time has to be manageable.
- Data should be random but free of outliers or uncontrolled noise so that results are reproducible and explainable.
- We need to be able to tweak certain properties of the data easily to observe its influence on the network.

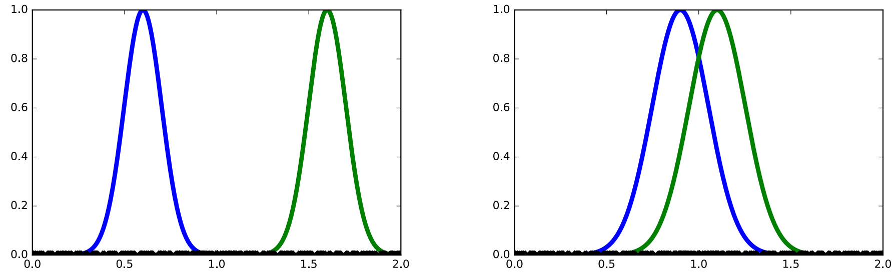
In order to create stationary data, we have to generate signals which are not localisation dependent but have a characterized spectral signature. Recall that when a graph filter h is applied to a GWSS signal, the result remains GWSS. Thus a simple way to artificially produce stationary signals is to filter white noise with a kernel of your choice. We chose to use non-normalized Gaussian bandpass filters of frequency band $[f_{min}, f_{max}]$, defined as:

$$
\begin{aligned}
h_{f_{min},f_{max}}(\lambda_l) &= \exp \frac{(\lambda_l - \mu)^2}{2\sigma^2}, \\
\mu &= \frac{f_{min} + f_{max}}{2}, \\
\sigma &= 0.3 \frac{(f_{max} - f_{min})}{\sqrt{2 \log{(2)}}}
\end{aligned}
$$

σ has been heuristically derived so that the width of the Gaussian matches with the frequency band $[f_{min}, f_{max}]$. We create two kernels having different frequency band and assign a label for each signal being filtered (i.e. $y = 0$ for a signal filtered with the first kernel, $y = 1$ for the second one).

The frequency band of the filter can be adjusted by choosing cutoff numbers $\{c_{min}, c_{max}\} \in [0, 1]$ such that $f_{min} = c_{min} * \lambda_{max}$ and $f_{max} = c_{max} * \lambda_{max}$, λ_{max} being the largest eigenvalue of the graph. This way we can easily control how "spaced" filters are, making the problem more or less difficult for the network. An example is displayed on figure 3, both figures display the PSD of the two stationary data generative kernels. On figure 3a, the band of the blue filter is set to $(c_{min}, c_{max}) = (0.2, 0.4)$ and the green filter one is set to $(c_{min}, c_{max}) = (0.7, 0.9)$, we can see that there is no overlap between filters and that bands are "narrow". On figure 3b, we brought the filters closer to each other and widened the bands.

[1] https://github.com/epfl-lts2/pygsp
[2] https://github.com/mdeff/cnn_graph



(a) Blue filter frequency band = [0.2, 0.4],
Green filter frequency band = [0.7, 0.9]

(b) Blue filter frequency band = [0.3, 0.6],
Green filter frequency band = [0.4, 0.7]

Figure 3: PSD of the filters with different frequency bands

After the filtering operation, white Gaussian noise with adjustable variance σ is added to make the problem more difficult. A flow chart of the data generation is displayed on figure 4.
We now have generated our dataset. We will take advantage of it to try to assess the problem to be optimized

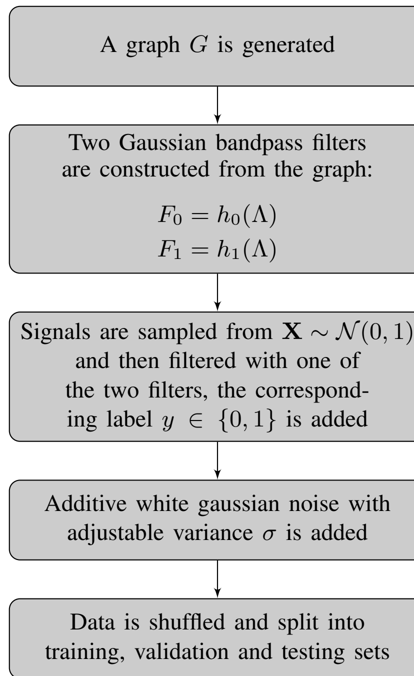


Figure 4: Data generation chart

4 Experiments

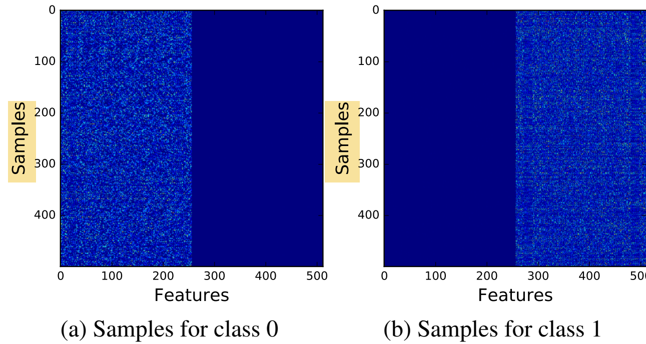
In this section we report our observations and results with different settings of our problem.

4.1 Assertion of the ground truth

We begin by checking our hypothesis that the optimal solution is the one for which the learned filters approximate the PSD of the generative filters. Consequently, since the filters are optimal and separate the frequency content, the columns of the fully connected layer weight matrix should have their sign inverted i.e. if there is a plus sign at row i for the first column, there should be a minus sign at row i of the second column, with approximately the same magnitude. Recall that the fully connected layer creates linear combinations of features learned in the convolutional layer. Thus with mutually exclusive features, the contribution of one particular feature should be increased for one class and decreased for the other, hence the inversion in signs for the same row.

To check this we filter again the samples with both filters and combine the outputs to create new samples of size $2 \times N$, then a non-linearity (a ReLU in this case) is applied. These two steps allow us to localize the energy content in certain features depending on the class labels i.e. for samples belonging to class 0, only the first N features will be active while the remaining ones will be set to 0 while for samples belonging to class 1 only the last N features will be active.

We fix the band of the first filter to $[0.2\lambda_{max}, 0.4\lambda_{max}]$ and the second filter to $[0.7\lambda_{max}, 0.9\lambda_{max}]$. A gaussian noise of variance σ^2 is added before the second filtering operation. Figure 5 shows the energy content of 500 samples of each class of the data matrix, here a grid graph of 256 nodes was used. We can observe that the filtering followed by the non-linearity "separated" the energy content of the data. Intuitively, those new features should be optimal as it creates a linearly separable space of the data. This preprocessing followed by the fully connected layer and the softmax function is similar to a simple logistic regression model.



(a) Samples for class 0 (b) Samples for class 1

Figure 5: Energy of the data after refiltering

We also fix the weights of the fully connected layer to be:

$$\mathbf{W} \in \mathbb{R}^{2N \times 2} = \begin{bmatrix} 1 & -1 \\ \vdots & \vdots \\ 1 & -1 \\ -1 & 1 \\ \vdots & \vdots \\ -1 & 1 \end{bmatrix} \begin{matrix} \left.\vphantom{\begin{matrix}1\\ \vdots\\ 1\end{matrix}}\right\} N \\ \left.\vphantom{\begin{matrix}1\\ \vdots\\ 1\end{matrix}}\right\} N \end{matrix}$$

We report here the mean test accuracy, averaged on 5 runs, with different noises. Noise here

is expressed in terms of the signal-to-noise ratio, defined as $SNR = \frac{P_{signal}}{P_{noise}}$ where P is the average power. Note that since both signals and noise are zero-mean, we used the alternative definition $SNR = \frac{\sigma^2_{signal}}{\sigma^2_{noise}}$.

SNR	Mean test accuracy (± std) (%)
20	100 (± 0)
2	100 (± 0)
0.4	99.88 (± 0.032)

Table 1: Ground truth test results

We can conclude that our hypothesis is correct and there exists an optimal solution that approximates the PSD of the generative filters which creates mutually exclusive features. We will now try to assess under what conditions does the network converge to this solution.

4.2 Fixing the Chebyshev coefficients of the filters

In this setup we initialize the learned filters with the coefficients extracted from the Chebyshev approximation of the generative filters and look at the fully connected layer. For reference, the average variance of graph signals is $\sigma^2_{signal} \simeq 0.07$ when the bands are [0.2, 0.4], [0.7, 0.9] and $\sigma^2_{signal} \simeq 0.2$ when the bands are [0.3, 0.6], [0.4, 0.7].

Results are displayed on table 2.

Filter bands	SNR	Mean test accuracy (± std) (%)
[0.2, 0.4], [0.7, 0.9]	20	100 (± 0)
	2	99.71 (± 0.41)
	0.4	98.87 (± 1.39)
[0.3, 0.6], [0.4, 0.7]	20	100 (± 0)
	2	99.91 (± 0.045)
	0.4	98.69 (± 0.736)

Table 2: Fixed filters test results.

As we can see the network gives near perfect predictions even with noise and high overlap between filters. Figure 6 shows the distribution of the weights in the fully connected layer after training, split in different blocks to exhibit the sign inversion that we hypothesized earlier. The difference in row indexes with the former case is due to implementation details. The Gaussian shape of the histogram comes from the Gaussian initialization.

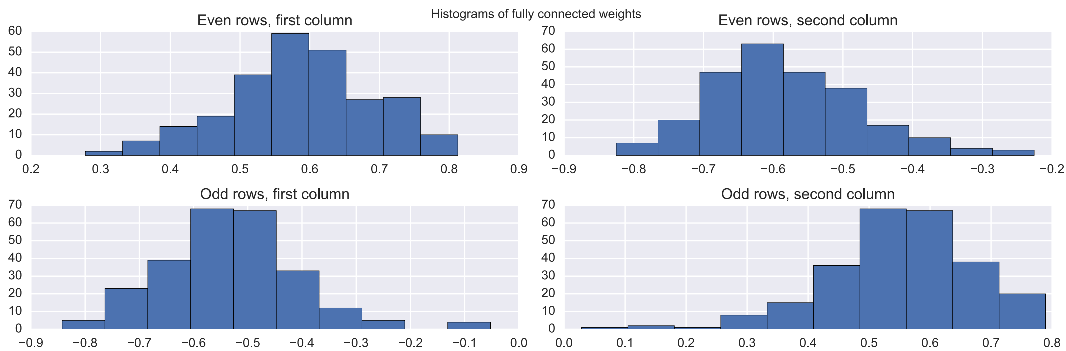


Figure 6: Histogram of weights of the fully connected layer after training.

4.3 Fixing the weights of the fully connected layer

We now let the network learn the filters but fix the weights of the fully connected layer to a coefficient α while keeping the optimal structure i.e. we set $\mathbf{W}$ to be :

$$\mathbf{W} = \begin{bmatrix} \alpha & -\alpha \\ -\alpha & \alpha \\ \vdots & \vdots \\ \alpha & -\alpha \end{bmatrix}$$

Figure 7 shows the PSDs of both the generative filters (top) and the learned filters after training (bottom) obtained for $\alpha = 1$, SNR=20 with a mean validation accuracy of 99.78%. Even though the network correctly predicted the classes, we see a clear discrepancy between the optimal solution and the obtained one. Only of the two filters is active, the amplitude of the red filter is very high and both are not smooth. With this setup the network is also very sensitive to initialization.

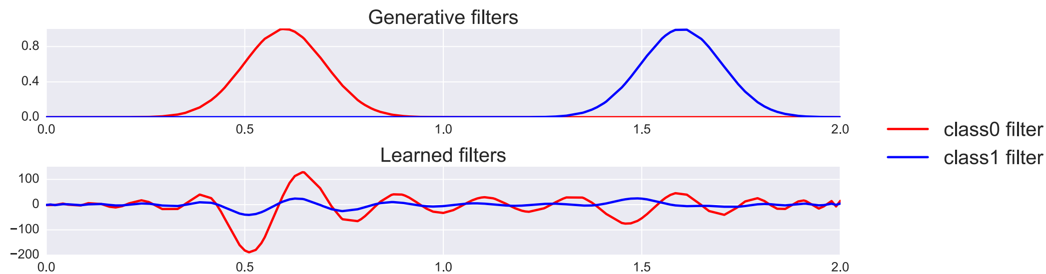


Figure 7: Results for $\alpha = 1$, SNR=20, with accuracy of 99.78%.

Figure 8 shows the PSDs of the filter for a network that hasn't converged. One interesting feature here is that the shapes of the filters are similar to the plot above but in this case the magnitudes are very high.

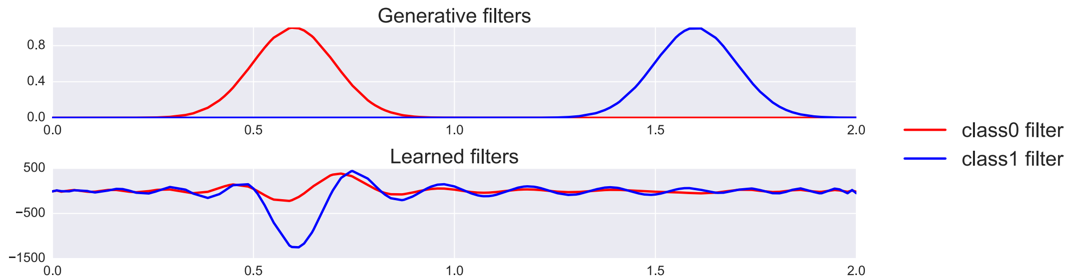


Figure 8: Results for $\alpha = 1$, SNR=20, with 50.80% of accuracy.

We can see that the results we obtained by fixing the fully connected layer diverged from the optimal solution. Other values of α were tried but gave similar results. Other noise variances were also explored. The qualitative difference between results with small noise and high noise is that convergence of the network occurs more often in the first case. Regarding the shape of the filters after convergence, we noticed similar results in both cases.

4.4 Tests with different initializations and regularizations

We begin by training the network without any constraints and see how "far" the result is from the optimal solution. Figure 9 shows the PSDs of the filters and the distribution of the fully connected weights obtained after training with 100% accuracy.

As we can see optimal structure of weight matrix of the fully connected layer is conserved but the learned filters do not closely approximate the PSD of the generative filters. Even though we notice a substantial improvement compared to the previous experiment. This result was consistent across several results that yielded perfect or near perfect accuracy. There can be several causes as to why we end up in local minima: the state space might be very large and thus sensitive to initialization or maybe the optimal solution can only be attained with additional constraints for the network.

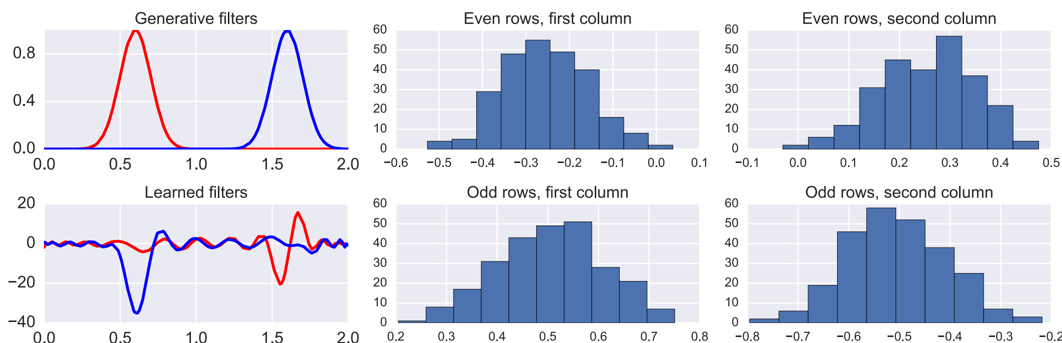


Figure 9: Results for an unconstrained model with 100% accuracy. (SNR=20)

Figure 10 displays the values of the Chebyshev coefficients learned during the training corresponding to figure 9. These scatter plots represent the value of each Chebyshev coefficient as a function of its order. During training an order $K = 30$ was used. The top row shows the Chebyshev coefficients extracted from the Chebyshev approximation of the generative filters while the bottom row shows the Chebyshev coefficients learned during training. We can notice a clear trend from the generative filters: there is a decay from low order coefficients to high order coefficients. This decay induces a smoothness of the approximated function and is non-existent in the trained filters. Thus a possible solution to help the network converge to the optimal solution would be to constraint the Chebyshev coefficients by adding an adaptive L2 regularization.

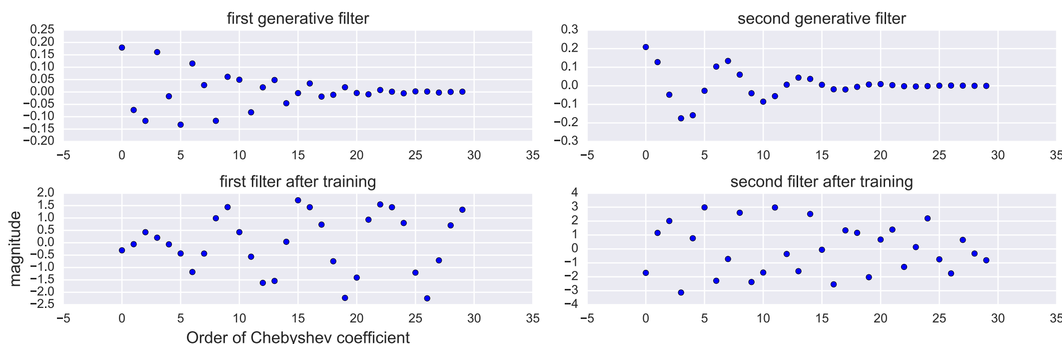


Figure 10: Scatter plots of Chebyshev coefficients.

Figure 11 shows the filters obtained with the coefficients regularization corresponding to the left hand side of the figure. With a mean accuracy of 99.74%, we can see that the PSDs of the filters obtained for this solution are closer to the optimal ones. Compared to the previous solutions, we now have a smaller magnitude for the filters, the shape is closer to a Gaussian and we reduced the ripple between the frequency bands. The issue here is that this local minimum is unstable: even though adding regularization makes the filters smoother with a smaller magnitude in general, most of the time results are not interpretable.
To cope with this, we can try different initializations of Chebyshev coefficients instead of the random normal one.

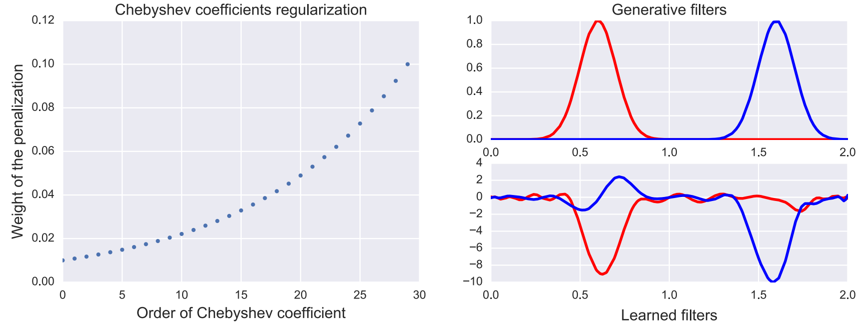


Figure 11: Results with regularization, SNR=20.

We begin with the "perfect" initialization i.e. with initialize the trainable coefficients to the ones of the generative filters but let them be eventually modified by the network, we also add a small regularization. Results are displayed on figure 12. As expected we have perfect accuracy and the shape of the filters is conserved. Still, we can see that the magnitude has been increased, even though the Chebyshev coefficients before and after training are highly similar.

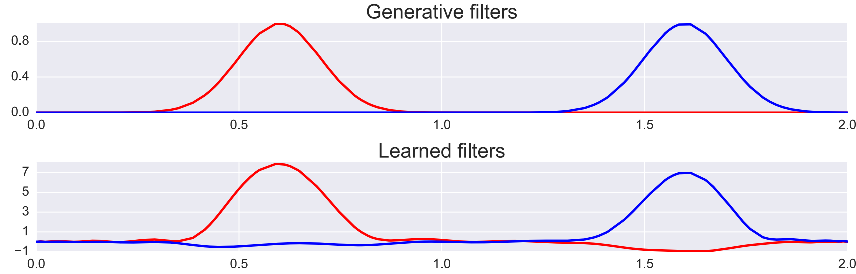
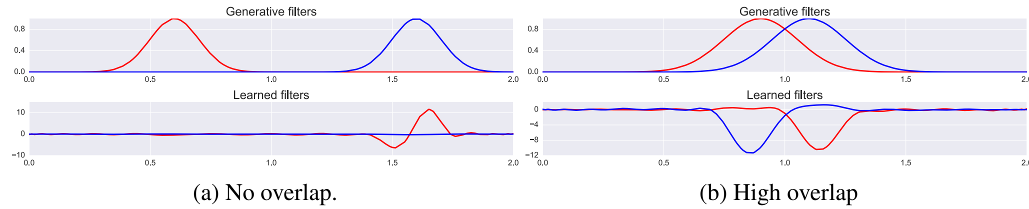


Figure 12: Results with initialization from the generative filters, SNR=20.

We now initialize all the coefficients to zeros, and add a small regularization. From figure 13a we can see that only one of the two filters is active, this makes sense since the filters do not overlap, activating for only of the two classes is enough to discriminate, which is verified by the fact that the network achieves 100% accuracy. From figure 13b we can see that the filters obtained by bringing the generative ones closer have a smooth shape. At the midpoint $\lambda \simeq 1$ the network has cut all frequencies to be able to discriminate.



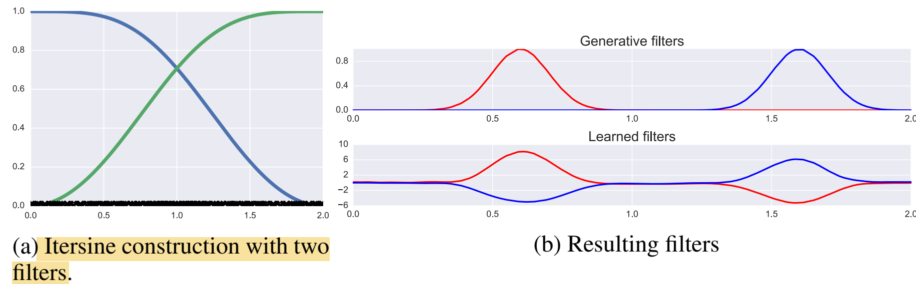
(a) No overlap.

(b) High overlap

Figure 13: Results with all zeros initialization, SNR=20.

Finally, we initialize the coefficients with an itersine[3] function. The PSD of the filter is shown on figure 14a. From figure 14b we can see that results are satisfactory, the learned filters are smooth because of the a priori smoothness of the initialization.

[3] `https://media.readthedocs.org/pdf/pygsp/latest/pygsp.pdf`, page 35



(a) Itersine construction with two filters.

(b) Resulting filters

Figure 14: Results with itersine initialization, SNR=20.

5 Conclusion

In this work we analyzed how a graph convolutional neural network works. We first reviewed the underlying theory and explained layer by layer what happens in the network.

With help of a synthetic dataset, we first verified our hypothesis that an optimal solution exists and how to characterize the solution. The experimental results showed how we can help the network converge to this optimal solution by adding different initializations and regularizations. This could eventually help us to better tune a network in a situation where we don't know the optimal solution but we have a prior knowledge on the form of the solution e.g. if we want to learn smooth filters.

The problem we characterized in this work can be considered as an easy problem. A possible future outlook would be to experiment with data where a point-wise non-linearity has been applied, hereby making the data non-stationary, and see if the network is able to make the data stationary after some non-linear transformations. A more complex data generation process and network structure would be involved.

References

[1] D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst, "Signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular data domains," *CoRR*, vol. abs/1211.0053, 2012.

[2] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *CoRR*, vol. abs/1606.09375, 2016.

[3] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016.

[4] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," *CoRR*, vol. abs/1605.05273, 2016.

[5] D. K. Hammond, P. Vandergheynst, and R. Gribonval, "Wavelets on Graphs via Spectral Graph Theory," *ArXiv e-prints*, Dec. 2009.

[6] N. Perraudin and P. Vandergheynst, "Stationary signal processing on graphs," *CoRR*, vol. abs/1601.02522, 2016.