Graphs for Machine Learning Project (MVA)
Convolutional Neural Network on Graphs

Alexandre Péré **Clément Tiennot**

Abstract

In this report, we present our understanding of the paper "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering" [4]. We both summarize the approach developed in the authors' work and provide an implementation in the `python` langage. Our implementation makes use of the `Tensorflow` package [1] which allows automatic computation of gradients and thus efficient optimization.

Introduction

Lately, Convolutional Neural Networks (CNN), have shown outstanding performances in classification tasks over a large range of data including images, audio and language. Those techniques are rapidly spreading across the scientific community, leading to unprecedented advances in different fields. Even though those performances are quite promising for future applications, the generalization of CNN to discrete signals over arbitrary (non-euclidean) structure is still an open problem. Indeed, it is worth noting that the data on which CNN give their best performances share an important common property: they all can be represented as contiguous data over regular lattices, like regular 2D arrays for images, or regular 1D arrays for sequences.

The convolution operation that confers their performances to CNN can be seen from two different viewpoints:

- The convolution operation is equivalent to a multiplication in the Fourier domain. This definition is not the most intuitive and is, to our knowledge, rarely used in practical implementations due to the cost of transferring data to the complex Fourier domain.
- The convolution operation is also equivalent to a classic filtering with a small sliding window, scanning and filtering the whole input signal (in the case of images, the filter can be a small 3x3 filter called kernel). This definition is much meaningful to understand the way CNN behaves, and is most often used for implementation.

From these two definitions, we understand that CNN performance relies heavily on the euclidean structure of the data, and the idea of locality and neighbourhood. Thus, it becomes clear that CNN can't be easily applied to data presenting less structure than images or sequences, for example in the case of signals defined over sensor fields, or trees.

The main question addressed by the paper is hence: How do we apply CNN to more general discrete signals over non-euclidean structures where straightforward convolution operation can not be defined. Graphs structure being the most general discrete structure we can define signals on, this question extends to: How do we apply CNN to signals defined over graphical structures?

In order to address this issue, we will have to rely on the Fourier Domain multiplication point of view. As we will see, one can use a spectral approach to define Fourier Transform over graphs, along with convolutions and filtering [3]. By choosing a specific filter parametrization (that uses the graph Laplacian) we can furthermore build localized and fast filters with performances comparable to classic CNN.

29th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

The Graph CNN implementation proposed in the paper is made possible by theoretic tools provided by the emerging field of Graph Signal Processing (GSP). This new field aims at studying discrete signals over non-euclidean structures, and is rapidly gaining attraction in the signal processing community. A complete introduction to the field can be found in [11].

1 A bit of Graph Theory

A weighted graph is defined as a tuple $(\mathcal{V}, \mathcal{E}, \mathcal{W})$, where $\mathcal{V}$ is a set of elements called Vertices (or Nodes), $\mathcal{E}$ is a set of couples of $\mathcal{V}$ elements called Edges (or Arcs), and $\mathcal{W} : \mathcal{E} \to \mathbb{R}$ is an application that assigns weights w to all edges.

Most of the time, the application $\mathcal{W}$ is represented by a *Weight matrix* W, which, if we consider a n nodes graph, is an $n \times n$ matrix. If an edge exists between two nodes $i, j \in \mathcal{V}$, then $w_{ij} \neq 0$, and if no edge exists then $w_{ij} = 0$. A graph can be directed or undirected. In our case we will exclusively use undirected graphs, meaning that the weights matrix is always symmetric. The *Degree* of a node i is defined as the sum of the weights related to it:

$$d_i = \sum_{k \in \mathcal{V}} w_{ik} \tag{1}$$

The *Degree matrix* D of a graph is a $n \times n$ diagonal matrix containing the degrees of all nodes i.

With these definitions we can introduce one of the most interesting tools used in graphs for machine learning, the graph *Laplacian matrix*, which is defined as:

$$L = D - W \tag{2}$$

This matrix is a fruitful representation of a graph that has a lot of interesting properties. In the case of Graph Fourier Transform that we will define in the next part, a slightly different definition of the Laplacian is used, i.e. the symmetric normalized Laplacian:

$$L_{sym} = D^{-1/2} L D^{-1/2} \tag{3}$$

Other definitions of the Laplacian can be found in the literature, having each different and interesting properties that we will not discuss here. For more insights on graph Laplacian and spectral clustering, one can refer to [12].

Last but not least, a discrete signal s over a graph $\mathcal{G}$ is defined as an application $s : \mathcal{V} \to \mathbb{R}$, that maps a real number to each graph nodes. In what follows, we will refer to such discrete signals over graphs, merely as signals.

2 Spectral filter on graph

In this section we introduce spectral filtering of signals defined over a graph.

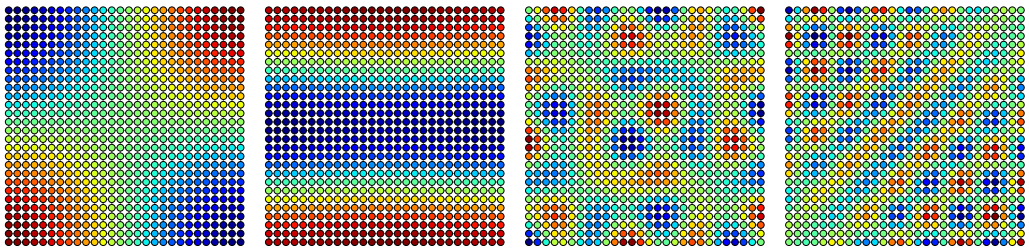
2.1 Spectral domain

For a graph $G = (V, E, W)$ with Laplacian $L \in \mathbb{R}^{n \times n}$, we have the following spectral decomposition, with $U \in \mathbb{R}^{n \times n}$ the change of basis matrix that diagonalizes the Laplacian L:

$$L = U \Lambda U^T \quad \text{with} \quad \Lambda = \mathrm{diag}([\lambda_0, \ldots, \lambda_{n-1}]) \tag{4}$$

The authors then introduce the Graph Fourier Transform (GFT) which takes a spatial one-dimensional signal (defined on the nodes of G) $x \in \mathbb{R}^n$ and maps it to its counterpart in the spectral domain $\hat{x} = U^T x \in \mathbb{R}^n$. The inverse operation $x = U\hat{x}$, allows to switch between the spatial on spectral domain at ease, provided the Fourier basis U is known.

On figure 1 we can see plots of some of the Laplacian eigenvectors over a regular 2D grid graph. We can see that those Fourier modes define complex periodic patterns, that shows quite clearly why the Laplacian eigendecomposition can be used as a Graph Fourier Transform.



(a) mode 91 (b) mode 215 (c) mode 474 (d) mode 613

Figure 1: Some Fourier Modes plotted over the graph (i.e. Laplacian eigenvectors)

2.2 Filtering

Applying a filter in the spatial domain amounts to perform an element-wise product in the spectral domain. Hence we can take a filter operator $g_\theta(.)$ that is defined in the spectral domain (and depends on some parameter vector θ) and apply it to $\bar{x}$: the output in the spectral domain is then $\bar{y} = g_\theta \bar{x} = g_\theta U^T x$. To get the final filtered signal y we go back to the spatial domain using the inverse GFT:

$$y = U\bar{y} = U g_\theta U^T x \tag{5}$$

In practice the element wise product of $U^T x$ by all the components of the filter g_θ can be achieved as a matrix multiplication if we define our filter as a diagonal matrix in $\mathbb{R}^{n \times n}$. It follows that any filter is of the form $\mathrm{diag}(\theta)$ with $\theta \in \mathbb{R}^n$. The authors refer to this general construction as a *non-parametric filter* because it has the maximum allowed degrees of freedom.

2.3 Localized filters

As pointed out by the authors, this general formulation is not satisfactory because it is not localized in space (each component of the output signal depends on every component of the input signal) and the number of parameters to learn is equal to the signal size which is a bottleneck and is not the case for standard convolutional filters on images.

One can show [6] that if the shortest path between two vertices i and j on the graph is strictly greater than K then $(L^K)_{i,j} = 0$. A natural idea that then arises is to design a filter that is parametrized over powers of the Laplacian (or in the spectral domain, the powers of Λ) to control the spatial localization of the filter:

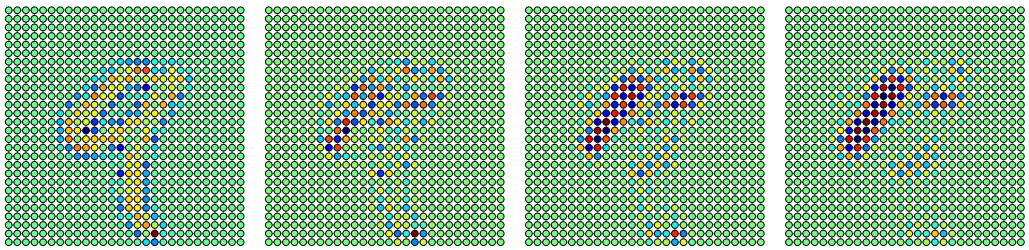
$$g_\theta(\Lambda) = \sum_{k=0}^{K-1} \theta_k \Lambda^k \tag{6}$$

Thus using the previously stated property it is easy to prove the spatial $K-$localization of such filters (see the original paper). In addition the learning complexity is now independent of the signal size since we only have K parameters.

Figure 2 shows an MNIST "9" sample filtered using different powers of the Laplacian. We can see that depending on K, only some large areas of stationary signals are kept, while the rest is discarded from the representation.

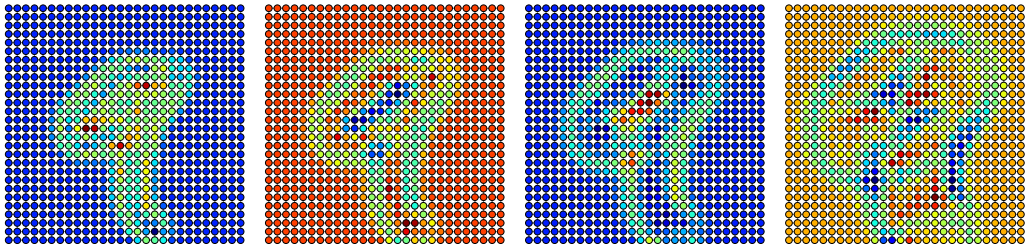
2.4 Fast filtering or Chebyshev approximated filters

Equation 7 is the theoretical key to construct a CNN on a graph, but for an efficient implementation it has one downside: in practice switching between the spatial and the spectral domain implies two matrix multiplications by U^T and U (which are dense matrices) and has a cost $\mathcal{O}(n^2)$. The authors, thus, make use of the Chebyshev polynomial approximation $T_k(x)$ defined recursively by $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$ with $T_0 = 1$ and $T_1 = x$ which forms an orthogonal basis of



(a) Λ^1 (b) Λ^3 (c) Λ^5 (d) Λ^9

Figure 2: Localized filtering of a signal using different powers of Λ.



(a) T_1 (b) T_2 (c) T_3 (d) T_5

Figure 3: Localized filtering of a signal using different orders of Chebyshev approximation.

$L^2([-1,1])$ with respect to the weights $1/\sqrt{1-x^2}$. The proposed filter is then defined as:

$$g_\theta(\Lambda) = \sum_{k=0}^{K-1} \theta_k T_k(\tilde{\Lambda}) \tag{7}$$

Where $\tilde{\Lambda} = 2\Lambda/\lambda_m ax - I_n$ is the scaled diagonalized Laplacian whose coefficients lie in $[-1,1]$. Because each T_k has degree k, this is a re-parametrization of equation 7 which proves directly the spatial localization. Finally the authors explicit the output signal y as $y = g_\theta(L)x = \sum_{k=0}^{K-1} \bar{x}_k$ with $\bar{x}_k = \theta_k T_k(\tilde{L})x$. The $\bar{x}_k$'s can be computed by the recursion $\bar{x}_k = 2\tilde{L}\bar{x}_{k-1} - \bar{x}_{k-2}$ with $\bar{x}_0 = x$ and $\bar{x}_1 = \tilde{L}x$. The filtered signal is then given by $y = [\bar{x}_0, \ldots, \bar{x}_{K-1}]\theta$ and the filtering cost is $\mathcal{O}(K|E|)$ because the matrix products during the recursion involve a sparse matrix (the scaled Laplacian) and dense vectors. Algorithm 1 summarizes how to perform the filtering. [1]

Algorithm 1: Fast localized spectral filter

Data: input signal $x \in \mathbb{R}^d$, filter parameters $\theta \in \mathbb{R}^K$, Graph Laplacian $L_\mathcal{G} \in \mathbb{R}^{d \times d}$
Result: filtered signal $y \in \mathbb{R}^d$
Initialization:
$\tilde{L} = 2L/\lambda_{max} - I_d$
$\bar{x}_0 = x$
$\bar{x}_1 = \tilde{L}x$
for $k = 2$ *to* $K-1$ **do**
  $\bar{x}_k = 2\tilde{L}\bar{x}_{k-1} - \bar{x}_{k-2}$
Return: $y = [\bar{x}_0, \ldots, \bar{x}_{K-1}]\theta$

[1] Our implementation doesn't benefit from the Laplacian sparsity since our goal was just to demonstrate our understanding of CNN on graphs and not to produce scalable code.

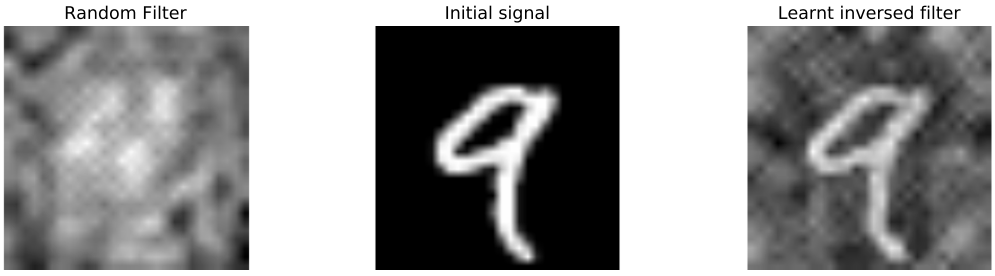


Figure 4: Left: filtered signal with $K = 100$ random coefficients. Center: initial signal on the grid graph. Right: filtering of the left signal after having learnt $K = 100$ coefficients to create an "inverse" filter using gradient descent.

On figure 3, we represented signals filtered using the recursive Chebyshev approximation. The behaviour here is different from the one we had with powers of Λ, and we can see that the difference formulation of the Chebyshev approximation tends to keep only the boundaries of the signal.

Learning the filters parameters can be achieved using back-propagation and gradient descent (see p.4 of [4] for the gradient expressions). Nevertheless, in our implementation, `Tensorflow` performs symbolic operations and is able to compute the parameter derivatives automatically which makes it really easy to implement the whole gradient descent even after cascading several filters.

An example of filtered signals is given by Figure 4: the graph is a grid where each pixel is a node and is connected to its 8 neighbors. On top of this graph the signal is the intensity of each pixel. We randomly chose $K = 100$ parameters for a filter and applied it to the signal using the Chebitchev formulation. As expected the filtered signal (left) is blurry. Just to demonstrate the gradient descent using `Tensorflow`, we tried to retrieve the initial signal (our "9" digit) from the filtered version to see if it is possible to learn an "inverse" filter. After 10 000 iterations of gradient descent, the mean squared error had fallen under 0.04 (i.e. about 20% average deviation since we normalized the intensity between 0 and 1). The reconstructed signal is far from perfect due to the loss of information during the first filtering but we clearly distinguish the shape of the "9".

3 Efficient Max-Pooling on Graphs

An important feature of CNNs is their ability to downscale the input information along the network, into a lower dimensional embedding that contains most of the semantic information present in the input. In classic CNNs, this dimension reduction results of two facts:

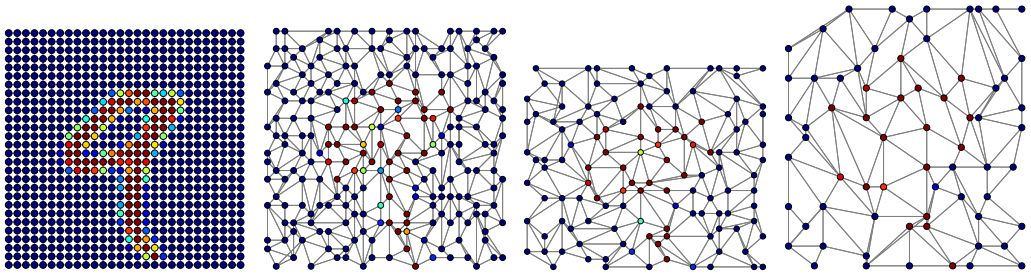
- The Max Pooling Operation: This operation slides a small window (2x2 pixels for images) over the whole input data, and keeps the maximum value in the window.
- The Convolution Operation: The convolution operation has the side effect to lower the dimension if the input is not zero padded.

Doing the same on a graph is a bit trickier; for example, the convolution operation used has no effects on the dimension, hence the importance to implement a Graph Max Pooling operation. The authors used a Graph coarsening routine from the Graclus multilevel clustering algorithm [5], to implement this behavior. They then describe an interesting way to re-index the graph nodes (adding extra "virtual" nodes) to allow performing all pooling operations as for a 1 dimensional signal.

3.1 Graph coarsening

The proposed method groups nodes by pairs, dividing the size of the graph by approximately two (some nodes can end up alone). It basically follows the steps:

- Randomly pick up one non-paired node and look at all its unpaired neighbors to find the one that maximizes a given local cut (the normalized cut $W_{ij}(1/d_i + 1/d_j)$ in our case).
- If for a node there are no remaining neighbors to pair it with, then we leave it alone.



(a) 784 nodes (b) 424 nodes (c) 229 nodes (d) 127 nodes

Figure 5: Example of pooled signal on four consecutive coarsened versions of an initial grid graph. The initial graph had $28 \times 28 = 784$ pixels/nodes. The spatial location of paired nodes was set to be the barycentre of the two initial node positions.

- Build a new graph where every found pair of nodes becomes a single node
- The weights are summed so that if (i_1, i_2) and (j_1, j_2) are two pairs of "old" nodes then the two new nodes i and j share an edge with weight $W_{ij} = W_{i_1 j_1} + W_{i_1 j_2} + W_{i_2 j_1} + W_{i_2 j_2}$.

A signal on the initial graph can be max-pooled by taking the maximum value of every pairs of nodes to assign it to the corresponding node in the coarsest graph.

Since the coarsening operation approximately reduces the size of the graph by two we can perform it twice to reduce the size by approximately four in a similar way to 2d max-pooling. The nested coarsened versions of the initial graph can be computed once and stored as their Laplacians along with some transition matrices that indicate how to pass from one graph to its coarsest (or backward, finest) version. An illustration of max-pooling using graph coarsening for one MNIST digit is given by Figure 5.

3.2 Nodes re-indexing to perform fast pooling

Even though this implementation using transition matrices is effective and allows us to decipher the pooled signals into graph representation, it is not particularly interesting in terms of computational performance (element wise matrix multiplication). To tackle this, we note that once we have obtained all the coarsened version of the initial graph (e.g. 4 or 5 of them), it is possible to reorder the nodes in a smart way that allows us to use a simple 1-D max pooling. We first assign arbitrary numbers to the nodes at the coarsest level. Then each node has either one or two parents and we add "virtual" nodes which will have 0 value (since we use a RELU activation after each convolutional layer they won't change the signal) so that each node has exactly two parents. We then number the nodes at the second coarsest level so that parents are side by side. We then repeat the process until we reach a numbering for nodes in the initial graph (and additional virtual nodes). It should be noted that adding a virtual node at one level implies adding 2 virtual nodes as its parents at the above level and then 4 virtual nodes to be the parents of the two parents and so on. Nevertheless in practice we don't need more than about 6 or 8 coarsened version of the graph so this stays manageable.

With the obtained numbering for the nodes of the initial graph, and after having added the virtual nodes, we can perform the pooling operation like a regular 1d pooling because two nodes to be aggregated together will always be side by side. In practice this is really convenient: the ordering just has to be computed once, we then just add the needed fake nodes (with 0 signal values) and rearrange the data. Then all max-pooling operation can be performed as if the signal was one-dimensional.

4 Putting it altogether: building and training a CNN on graphs

In order to test our understanding of the algorithms and to gain intuition about the potential of GCNN on classic Neural Network tasks, we have implemented the different algorithms presented in the

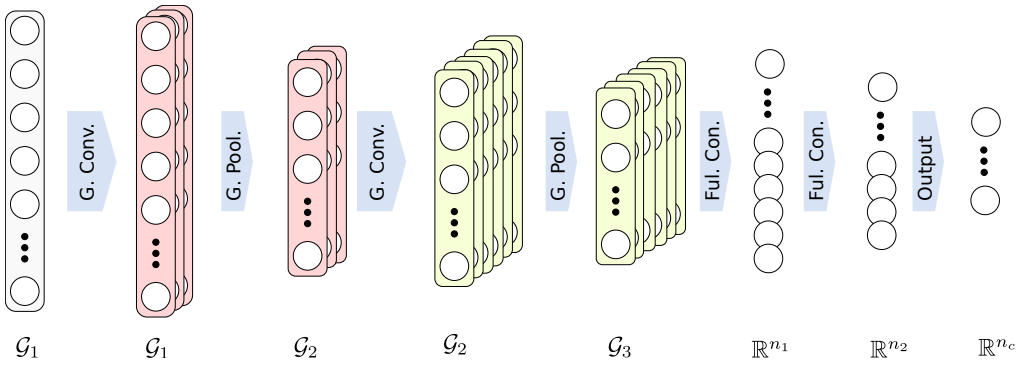


Figure 6: The architecture used to classify MNIST, with signal structure

paper. When coming to Neural Networks, two experiments allow to rapidly gain intuition about the potential of an architecture:

- Classification: Check that the architecture proposed is able to classify a simple data-set with acceptable performance.
- AutoEncoders : Construct an encoder-decoder architecture to verify that the model allows to construct interesting embedded representation of simple data.

Following the work of the authors, we decided to implement both experiments using a classic dataset: the MNIST dataset of handwritten digits [10]. This dataset has multiples advantages: it is reasonable in size, it is a widely used benchmark on which a vast majority of algorithms have been tested, and it uses images which helps to rapidly debug implementation (we can basically plot and check everything). Still, using MNIST has a few downsides we would like the reader to be aware of: the samples are highly pre processed (centered, resized, and upright), which makes it easier to classify, and more specifically to our case, images are not the best examples of signals defined over *arbitrary* structures (like data over sensing networks).

4.1 A classic benchmark: The MNIST classifier

To implement a full GCNN classifier over the MNIST dataset, we took the approach to implement it in two times. First, we implemented a slow but easier to debug program using equation 7 for filtering, and Transition matrix based Max Pooling. For this first implementation, we have tried different approaches based on the classic LeNet5 architecture [9], to finally come up with our best working one. Our basic Graph Convolutional stack is hence composed of the following operations:

- Perform a Graphical filtering with largest Laplacian power $K_i = 5$, and number of filters $n_i = 5$.
- Perform a ReLu Non-Linearity
- Perform 2 Max Pooling to reduce the overall size about a factor ~ 4

Note that we chose to apply every filter to every feature independently, which means that the last feature map will have a depth equal to $\prod_i n_i$.

Our first general architecture is then a combination of two such Graph Convolutional stacks, followed by 2 fully connected layers and a softmax output layer, as represented on figure 6.

We trained this architecture using a classic regularized cross entropy loss:

$$\mathcal{L}(x, \hat{y}, \theta) = -\sum_{i=1}^{n} \hat{y} \log P(y|x, \theta) + \alpha \sum_{j=1}^{m} \|\theta_j\|^2 \tag{8}$$

With a dataset of n samples $\{(x_i, \hat{y_i})\}$, m parameters θ_j, and a regularization hyper-parameter $\alpha = 0.001$. We optimized this loss over the parameters using an AdaDelta optimizer [13], on mini-batch gradient descent (number of samples per batch = 300). On figure 7, we can see the training

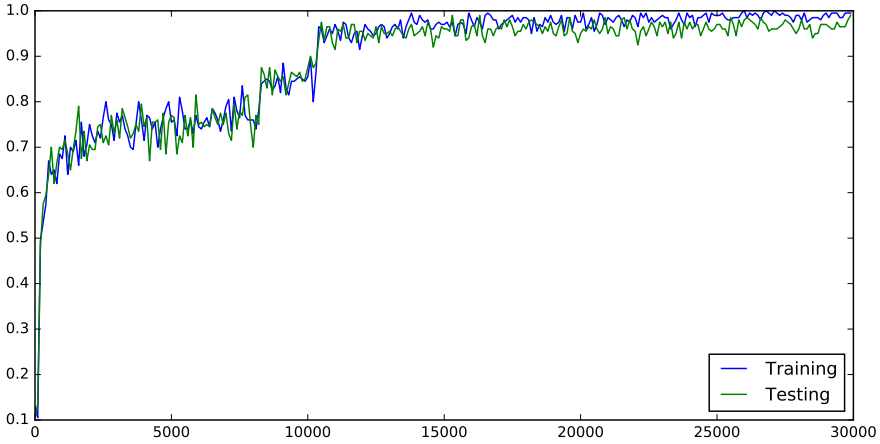


Figure 7: Test and Train Accuracy through epochs. Final Test Accuracy: 96.9%

Implementation	Epoch duration
Graph Fourier Transform & Transfert Matrix Pooling	0.29s
Chebyshev Approximation & 1-D Max Pooling	0.10s

Table 1: Training epoch duration using slow and fast implementations, on GTX1080. The fast implementation allows a speedup of $\sim \times 3$.

curve and the best test accuracy obtained of $\sim 96.9\%$. We stress out that in this work, the objective was to check that GCNN are able to learn from data, and not to perform a comparative study of the best GCNN architecture possible. Consequently, the performances of this classifier are not to be compared directly with other CNN performances, as no extensive tuning of architecture has been made to achieve the best possible accuracy. Moreover, it is worth noting that in the case of the classic LeNet5 architectures, each filter is of size $5 \times 5 = 25$, which is five times the length of the filters we used. To sum up, this test accuracy shows that the network has effectively learnt to classify the data set, and the tuning of the architecture could be subject to further work to obtain performances similar to the ones reached by classic architectures.

In a second time, once the principles of our implementation were validated by the first experiment, we have reimplemented our architecture using the two faster operations of Chebyshev Approximated Filters, and 1-D Max Pooling. Using this implementation allowed us to reach similar performance but in much shorter time as we can see on table 1.

4.2 Going Further with Autoencoding

To push further the experiments already made in the paper, and to meet the final goal of this project, we have implemented a simple single filter autoencoder architecture, that we have trained in an unsupervised manner over the MNIST data. For the encoding part, we used the following Graph Convolutional stack:

- Perform Graphical Filtering with largest Laplacian power $K_i = 5$, and number of filters $n_i = 1$
- Perform a ReLu Non-Linearity
- Perform a single Max Pooling to reduce the overall size about a factor ~ 2.

Concerning the decoder part, the data needs to be projected back to the larger graph at each stack. This is performed by simply projecting the signal values at coarsened nodes on their non-coarsened parents. This operation was easily implemented using the Transition matrix. Our Decoding Graph Convolutional stack was thus made of the following operations:

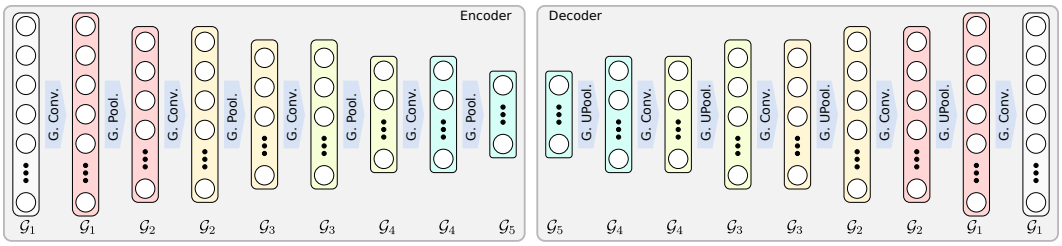
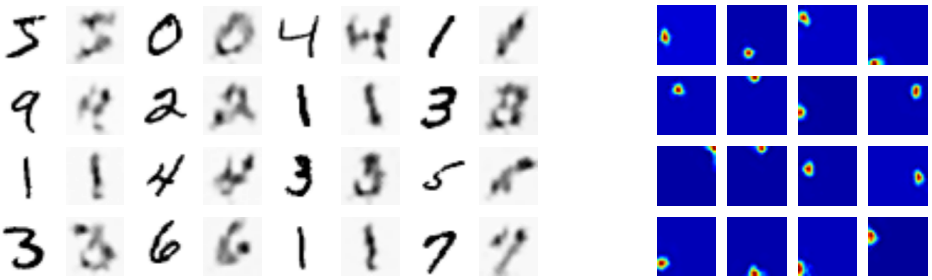


Figure 8: The architecture used to autoencode the MNIST DataSet, with signal structure.



(a) Reconstructed samples

(b) Code projection to output space

Figure 9: Reconstructed samples, and code variables projected to the output space

- Perform 2 Max UnPooling to enlarge the data of ~ 2
- Perform a ReLu Non-Linearity
- Perform a Graphical filtering with largest Laplacian power $K_i = 5$, and number of filters $n_i = 1$.

The overall architecture is composed of four Encoding Graph Convolutional stacks , and four Decoding Graph Convolutional stacks as depicted on figure 8. We note that the embedded code is a vector of length ~ 60 depending on the coarsening operation (compared to the input size 784).

We trained this architecture using a regularized L2-Distance:

$$\mathcal{L}(x, \theta) = \|x - f(x)\|_2 + \alpha \sum_{j=1}^{m} \|\theta_j\|^2 \tag{9}$$

With $f(x)$, the output of the network for the training sample x. We trained the whole architecture from scratch, using again the AdaDelta Optimizer.

Using our trained autoencoder we were able to partially reconstruct samples as depicted on figure 9. While keeping the overall shape of the digit, the reconstructions are still quite blurry, and our experiments have shown that the deeper the encoding, the blurrier the reconstruction was. To get deeper understanding of the learned embedded structure, we decoded some variable indicators of the embedded code (decode a $[0, 0, 1, 0, 0, \ldots, 0]$-like vector). When looking at the decoded plots of the first 16 indicators, we can see that the autoencoder has basically learnt to project the code signal from coarsened graph nodes to its parents nodes on the output space. In other words, it has learnt to approximate the identity operation. To our knowledge, it is a common pitfall in autoencoders, which can be overtaken using more complex forms of training (denoising for example).

By the end of our experiments with autoencoders, it was still unclear if better behavior could be obtained using GCNN models. Once again, further work might be necessary to gain intuition about the potential of GCNN for autoencoding.

5 Conclusion

This project allowed us to work on the cutting edge topic of non-euclidean CNN. Interestingly, the ideas developed here can easily be extended to be used on data defined over Manifolds, as suggested in [2], which could lead to even larger applications. It is worth noting that since the beginning of this project, two unpublished (and not reviewed by us) papers have been proposed on the topic of GCNN and GCNN Autoencoders, [8] [7]. The subject is hence in active research state, which explains why few experiments were made on the best architecture existing to tackle general Neural Networks problems. Hence the results produced during this project should be interpreted regarding the relative youth of those techniques, that do not benefit from a large research corpus as classic CNNs do. Nevertheless, regarding the results we had in our experiments, we are quite confident that great achievements will be made on this subject in a very near future.

We would like to thank our adviser and reviewer *Florian Strub*, for giving us the chance to work on such an interesting subject, and for his help to tackle the `tensorflow` implementation. The code we produced as part of this project is available at `https://github.com/ctiennot/graphConvNet`.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: going beyond euclidean data. *CoRR*, abs/1611.08097, 2016.

[3] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

[4] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3837–3845, 2016.

[5] I. S. Dhillon, Y. Guan, and B. Kulis. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE transactions on pattern analysis and machine intelligence*, 29(11):1944–1957, 2007.

[6] D. K. Hammond, P. Vandergheynst, and R. Gribonval. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2):129–150, 2011.

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

[8] T. N. Kipf and M. Welling. Variational Graph Auto-Encoders. *ArXiv e-prints*, Nov. 2016.

[9] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2323, 1998.

[10] Y. LeCun, C. Cortes, and C. J. Burges. The mnist database of handwritten digits, 1998.

[11] D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst. The emerging field of signal processing on graphs. *IEEE Signal Processing Magazine*, 30(3):83–98, 2013.

[12] U. Von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, 2007.

[13] M. D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv*, page 6, 2012.