Combining graph neural networks and classical kernels for graph generation

Eran Shmuel (301796561), Ethan Fetaya

Submitted as final project report

1 Introduction

Generative Adversarial Network[7] (GAN) can be a useful approach when trying to generate new images, videos, audio and graphs. In this project our goal is to improve the traditional GAN approach for building graphs by using the MMD loss function with multiple kernels. A traditional GAN approach for graphs, which use a graph neural network (GNN) as the GAN's discriminator has the same limitations that GNN have, and can not distinguish between all possible graphs. Our approach is to use a GNN as a learned kernel, but combine it with other kernel functions, like the Laplacian kernel, and the shapes kernel in order to utilize the advantages of GNN, but not to rely on it alone.

Generative Moment Matching Networks[14] (GMMN) is an approach to generate images which use the kernel function as a closed-form discriminator. The generator is trained to minimize the Maximum Mean Discrepancy (MMD), which is the distance computed between the real and fake data, using the kernel function. The approach was good in theory but does not work well in practice, relative to modern GAN approaches. In a follow-up work, called MMD-GAN[4] the authors proposed to learn the kernel. The main problem with the GMMN approach was that it used a fixed kernel, which might not good enough when using a finite Monte-Carlo approximation because the MMD function is not accurate due to the use of finite data. The learned kernel, also called MMD GAN, is an approach similar to the GMMN, but instead of applying the kernel function k directly on the data, it uses a neural network F as an injective function and applying the kernel $k' = k \circ F$. The neural network F is trained to maximize the MMD distance between fake and real data, which makes the kernel k' more informative.

While MMD GAN did perform better then GMMN, it is still outperformed by modern GAN approaches for generating images. One thing GNN GAN proved is that using more informative kernels can improve the performance of the generator. We believe that unlike images, there are good informative kernels functions to compute similarity measure between graphs.

Molecular GAN[11] (MolGAN) is another approach for generating molecular graphs by using a traditional GAN approach adapted to discrete graph structure. It was further extended with a reinforcement based reward function. It uses a multi layer perception as a generator and a Graph Convolutional Network[10] (GCN, variant of GNN), as a discriminator. We decided to build our models with similar architecture to to MolGAN, because we wanted to test the influence of changing the loss objective and not the influence of changing the models. We used the same generator, and used the GCN model as the learned kernel.

We tested this project on the QM9 chemical database, and compared the results to MolGAN. While the MolGAN paper tested the valid score, which is the percentage of valid graphs created by the generator, we thought that this method is flawed because a generator which makes the same valid graph all the time have the maximum valid score. Instead we chose to look at the valid-unique score, which is the percentage of valid and unique graphs created by the generator. We were able to improve the peak valid-unique score of the model by 8% more then the results of MolGAN, even through we didn't use the MolGAN's reward function which is used to optimize the model on an approximation of the valid score.

2 Background

2.1 GAN

Generative Adversarial Networks (GAN) is an approach in the deep learning field[7] which uses two neural networks in order to generate novelty data. Given data X where $X \sim \mathbb{P}_d$ The framework goal is to generate objects from $\mathbb{P}_\phi \approx \mathbb{P}_d$ The framework contains two neural networks: the generator and the discriminator. The goal of the generator is to create objects that fool the discriminator, and the goal of the discriminator is to distinguish between real objects and the fake objects that were created by the generator. The objects can represent images, videos, audio files graphs, and more. The generator gets an input z from a base distribution $\mathbb{P}_z$ such as Gaussian distribution or uniform distribution, and tries to minimize the function $Log(1–D(G(z)))$. The discriminator aims to maximize the function $Log(D(x)) + Log(1–D(G(z)))$. Together, the two networks objective is optimize the minmax function:

$$\min_G \max_D E_{x\sim\mathbb{P}_d}(log(D(x))) + E_{z\sim\mathbb{P}_z}(log(1 - D(G(z)))$$

In practice, the generator doesn't get effective gradients during training, because the derivative of $Log(1 - D(G(z)))$ is low when $D(G(z))$ returns values close to 0. In order to fix it, the generator is trained to maximize $log(D(G(z)))$ instead of minimize $Log(1 - D(G(z)))$

2.2 GMMN

Generative Moment Matching Networks (GMMN)[14] is an approach similar to GAN in spirit, that using a closed-form discriminator using a Reproducing kernel Hilbert space(RKHS). Instead of using a discriminator to distinguish between objects came from P_ϕ and objects came from $\mathbb{P}_d$, the generator uses a loss function called Maximum Mean Discrepancy (MMD) which it tries to minimize. The loss function objective is to compute the approximated distance between P_ϕ and $\mathbb{P}_d$. Let H be a reproducing kernel Hilbert space, and Let $\mathcal{F}$ be a set of functions from H that defined as $\mathcal{F} = \{f \in H : ||f||_H \leq 1\}$, MMD is defined by:

$$MMD(\mathbb{P}_d, P_\phi) = \sup_{f \in \mathcal{F}} |E_{x \sim \mathbb{P}_d}(f(x)) - E_{y \sim \mathbb{P}_\phi}(f(y))|^2$$

One can show that the suprimum has a closed form solution here as

$$MMD(\mathbb{P}_d, P_\phi) = E_{x \sim \mathbb{P}_d, x' \sim \mathbb{P}_d}(k(x, x')) + E_{y \sim \mathbb{P}_\phi, y' \sim \mathbb{P}_\phi}(k(y, y')) -$$
$$-2E_{x \sim \mathbb{P}_d, y \sim \mathbb{P}_\phi}(k, y))$$

The MMD loss function is the Monte-Carlo estimation of the MMD distance using N samples

$$L_{MMD} = \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} k(x_i, x_j) + \frac{1}{M^2}\sum_{i=1}^{M}\sum_{j=1}^{M} k(y_i, y_j) - \frac{2}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M} k(x_i, y_j)$$

While the GMMN idea seems promising in theory, in practice it didn't perform well on most of the images data-sets. MMD GAN[4] is an extension which uses the MMD loss function. The main problem with the GMMN approach is that the kernel might not be good enough to distinguish between $\mathbb{P}_d$ and $\mathbb{P}_\phi$. Because the MMD loss function is computed on finite items, even if it is applied on two groups of items from the same distribution the loss will almost always be more then 0. The loss can also be very close to 0 when it's computed on two different distribution. For example, when using the Gaussian kernel, choosing a very big bandwidth parameter will result of the distance being always 1, while choosing a very small bandwidth parameter will result with the distance being always 0, besides when applying the kernel on the same data point. Because bad kernels function can result with no informative result, it is very important to choose a good kernel function. In order to choose the best kernel for our problem we can choose the kernel which maximize the MMD distance between $\mathbb{P}_d$ and $\mathbb{P}_\phi$, which will make him more likely to distinguish between the two distributions.

$$\max_k MMD_k(\mathbb{P}_d, \mathbb{P}_\phi)$$

But choosing a kernel which maximize the MMD distance can be extremely hard mission. The proposed solution was to use a learned kernel. Instead of finding the best kernel between characteristic kernels, we can use 1 kernel, and an injective function F and define the kernel as $k' = k \circ F$. We can now find the function F which maximize the MMD distance

$$\max_F MMD_{k \circ F}(\mathbb{P}_d, \mathbb{P}_\phi)$$

This can be done by choosing F to be a neural network. Together with the generator loss, we can define the framework objective to optimize the following minmax function:

$$\min_{\phi} \max_{F} MMD_{k \circ F}(\mathbb{P}_d, \mathbb{P}_\phi)$$

The minmax objective is very similar to the GAN framework objective, hence the name MMD GAN. The authors of the MMD GAN paper also decided to force the F function to be reversible, which can be done by training it together with a decoder F^{-1}, and train the decoder on objects from both distributions. While in theory this could make the learned kernel training more stable, in practice the authors reported that it didn't help the training.

2.3 Graph neural network

Graph Neural Network (GNN), is a neural network performed on graphs of the form of (A,V) where A is the adjacency matrix, and V are the graph nodes, each is represented by a set of features. The network contain multiple layers which aggregates over each node's neighbors features and pass them to the next layer.

$$F_t(v) = F_{merge}(F_{t-1}(v), F_{aggr}(f_{t-1}(u) | u \in neighbors(v)))$$

Where F_{aggr} is a differentiable, permutation invariant neural network used to aggregate over the neighbors features and f_{merge} is a function used to merge between the node representation and his neighbors representation by concatenation or by a more complicated neural network. A basic GNN will use $F_{aggr}(X) = \sum_{i=1}^{n} x[i]$ and $F_{merge}(a, b) = \sigma(Aa + Bb)$ where σ is a none linear function like Sigmoid, Tanh or Relu, and A, B are linear mappings.

$$F_t(v) = \sigma\left(AF_{t-1}(v) + B \sum_{u \in neighbors(v)} (f_{t-1}(u))\right)$$

The network compute a vector representation of every node in the graph, and output the sum of them in order to compute the vector representation of the graph. GNN have limited expressive power, research proved that it can't distinguish between every two graphs that passes 1-dimensional Weisfeiler-Leman graph isomorphism heuristic[3].

2.4 Related Works

Molecular GAN (MolGAN) is a Generative Adversarial Network made to create new molecular graphs[11]. The network create only small graphs with bond types, learned from the QM9 data-set. It uses a very similar approach to GAN, but also adds a reward function network R. The network's generator objective is to fool the discriminator, and also maximize the reward. The reward function objective is to calculate approximation of various scores on the generated graphs. It is computed with reinforcement learning[12] based on the real reward provided by an external system. Because the generator outputs discrete values, before it passes the generated graph to the discriminator, the network perform gumbel softmax[5]. The network uses a permutation invariant

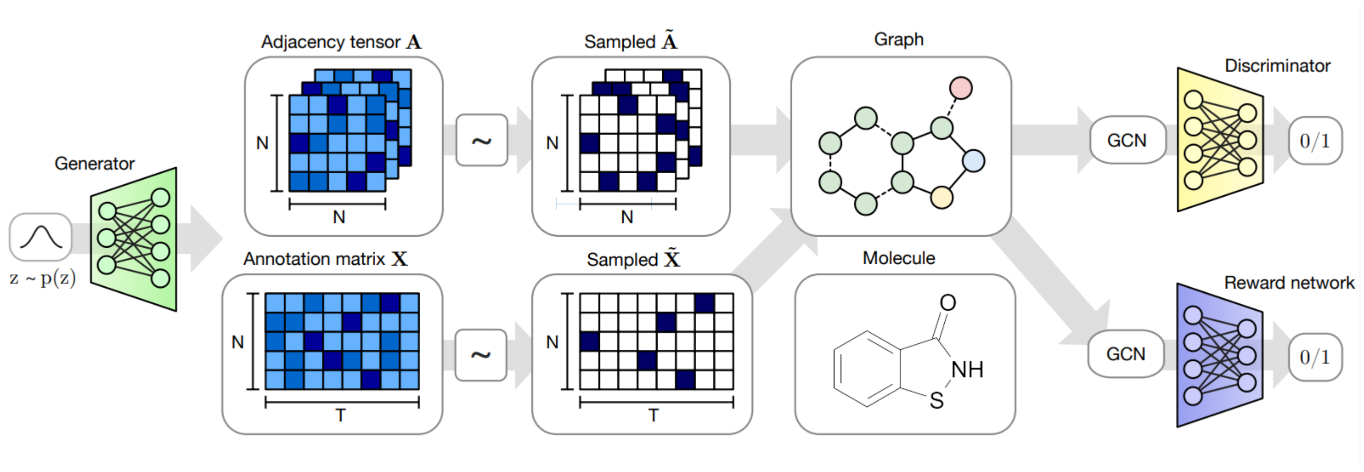

Figure 1: The MolGAN aproach, from the MolGAN paper[11]

Relational Graph Convolutional Network (GCN)[10] as a discriminator and for the reward network, which is a variant of GNN made for graphs with multiple edges type. The network tries to optimize Wasserstein GAN(WGAN) loss function[9], which is a variation of the GAN loss function. The discriminator is trained with gradient penalty[8], which is improvement of the gradient clipping scheme used in WGAN. Together with the reward function, the generator tries to maximize the function:

$$\lambda(D(G(z))) + (1 - \lambda)(R(G(Z)))$$

The discriminator is trained to minimize the function:

$$-D(x) + D(G(z)) + \alpha(|\Delta_{\hat{x}} D(\hat{x})| - 1)^2$$

Where $\hat{x}$ is a linear combination of x, G(z), and α is the weight given to the gradient penalty.

Objective-Reinforced Generative Adversarial Networks (ORGAN)[6] is another approach for generating molecules which is similar to Molgan because it's also trying to maximize a reward function. The big different is that it's creating molecules which are represented by SMILES (strings generated from the molecules graphs) with LSTM based generator. Other methods to generate graph include using auto encoder for edge prediction within graph[14], using a likelihood based method to output a graph in a sequential manner[13], using variational Autoencoders (VAE)[1] and combination auto-regressive and flow-based approaches[2].

3 Solution

3.1 General approach

Generative models for graph structured data can be very useful for creating molecule structures, social networks, probabilistic models and more. Modern models are built with a very complicated minimax loss function. Our approach is to use the maximum mean discrepancy (MMD) and multiple kernel functions to create a more efficient loss function, which can help us increasing the quality of the generated graphs. The MMD was used before to generate images, and while it did create images successfully, the quality of the images wasn't good enough. We believe that it's because there aren't good kernel functions to compute similarity measure between images. MMD GAN proved that using a better kernel can help increase the quality of the images, which made us believe that informative kernels like the Laplacian kernel and the Shapes kernel can be very effective when trying to use the MMD loss function when generating graphs. Those kernel can distinguish between graphs that GNN can't. Research proved that GNN can't distinguish between every two graphs that passes 1-dimensional Weisfeiler-Lemangraph isomorphism heuristic, that prove that GNN can't distinguish between a graph with only 2 triangles and a graph with only a hexagon. Both the Laplacian and the Shapes kernels can distinguish between those graphs. The Shapes kernel can do it simply by counting the number of triangles. The Laplacian kernel can distinguish between the two graphs by look at the number of connected components, which he can calculate by counting the number of zeroes in the eigenvalues we got from the symmetrically normalized Laplacian matrix.

3.1.1 GCN kernel

Our main approach was to use GNN as the main kernel, and combine it with other kernels like the Laplacian kernel and the shapes kernel. The main kernel is essential, because the other kernels doesn't look at the bond types between nodes, and doesn't look at the values of each node. We tried multiple ideas for the main kernel:

1. A constant GCN with random parameters
2. Randomize the GCN parameters every x iterations
3. Using a learned kernel
4. Using multiple GNN kernels. This can be done by using multiple kernels of the same type, or by using multiple kernels from different types.

3.1.2 Laplacian kernel

The Laplacian matrix can be very informative when looking at simple graphs and weighted graphs. Its defined as $L = D - A$ where D is the degree matrix and A is the adjacency matrix. In our case, we wanted to use the the symmetrically normalized Laplacian matrix which is defined as $L_{sym} = D^{\frac{1}{2}} L D^{\frac{1}{2}}$, because we wanted the matrix to be positive definite. In order to compute the symmetrically normalized Laplacian matrix, we first had to replace the bond type of the graphs with a boolean value that says if there is a bond between nodes or not. We then computed the eigenvalues of the symmetrically normalized Laplacian matrix, and sort them. At the end, we computed the MMD distance between the eigenvalues we got from the real graphs and the fake graph.

3.1.3 Shapes kernel

The shapes kernel is another informative kernel when looking at simple graphs. Its objective is to find the similarities between the number of polygons of each graph. In our case, we looked at the number of triangles, squares, pentagons, hexagons that started from each node of the graph. In order to compute the number of triangles started from a node at a graph, we can look at the matrix A^3, where A is the adjacency matrix. For each i node of the graph, the numbers of triangles starts from that node is $A^3[i][i]$. In order to extract features from the whole graph, we looked at the mean and standard deviation of the diagonal of A^3. In a similar way we can look at the diagonal of A^N for every polygon with N nodes and compute the mean and standard deviation. At the end we computed the MMD distance between the polygons normalized features extracted from fake and real graphs.

3.1.4 The Gaussian kernel

We computed all the MMD distances with the Gaussian kernel $k(x, x') = exp(-\frac{1}{2a}|x-x'|^2)$. One important thing when computing the Gaussian kernel is to have a good value for the bandwidth parameter. Bad values can results with the distance between different features to be always 0 or 1, which give us no information about the true distance between the 2 sets. We tried 4 approaches for choosing the The bandwidth parameter

1. Use the the bandwidth parameter which gives the highest variance when computing the kernel between every 2 sets of features from the real graphs at the start of the training.
2. Do the same as (1), but for every x iterations find the best bandwidth parameter between half of the current bandwidth parameter, and twice the bandwidth parameter
3. Start with the bandwidth parameter like in (1), and use it as a learned parameter of the learned kernel
4. Use multiple bandwidth parameters which centered around the value computed at (1) and return the sum of the distances.

3.2 Design

3.2.1 The generator architecture

We used the same generator which was used in the MolGAN paper. the generator consist of 3 linear layers with the Tanh activation function and an optional dropout, followed by 2 linear layers, one that outputs the graph's edges and one that outputs the nodes.

3.2.2 The learned kernel architecture

We used the same architecture for the learned kernel as the architecture that was used for the discriminator in the MolGAN paper, but instead of using the final scalar from the network, we are getting the features before the last linear layer, and use them to compute the MMD distance. The network is a relational GCN[10], which is a convolutional network for graphs with multiple edge types. The GCN starts with multiple graph convolutional layers defined as:

$$h_i^{(l+1)} = tanh(f_s^{(l)}(h_i^{(l)}) + \sum_{j \in N_i} \sum_{e=1}^{E} \frac{A_i je}{N_i} f_e^{(l)}(h_j^{(l)}))$$

Where h_0 is the graph's nodes, A is the adjacency matrix, N_i is the neighbors of the node i, and E is the number of edge types. $f_s^{(l)}$ is a linear function for the current node, and $f_e^{(l)}$ is a linear function that we use for each edge type. The convolutional layers are followed by an aggregate layer, defined as:

$$agg_{out} = \sum_{v \in V} \sigma(F_1(h_o)) \otimes tanh(F_2(h_o))$$

where h_{out} is the output of the last convolutional layer, f_1, f_2 are MLP and σ is the sigmoid function, $\otimes$ is a element wise multiplication. The network then output $tanh(F_3(agg_out))$, where F_3 is another MLP.

3.2.3 The loss functions

The generator aims to minimze the following loss function:

$$\frac{\sum_{d \in D} MMD(d(G(z)), d(x))}{|D|} + L * MMD(Laplacian(G(z), Laplacian(x)))+$$
$$+S * MMD(Shapes(G(z), Shapes(x)))$$

Where D is the group of learned kernels, Laplacian is a function that get a graph and return the eigenvalues of the graph's symmetrically normalized Laplacian matrix, Shapes is a function that gets a graph and return the mean and standard deviation of 3 to 7 length polygons. L and S are hyper-parameters.

Every leaned kernel d aims to maximize the following loss function:

$$MMD(d(G(z)), x(d))$$

We also wanted to test the influences of training the Laplacian and the Shapes kernel, in this case, the generator aims to minimize the function:

$$\frac{\sum_{d \in D} MMD(d(G(z)), d(x))}{|D|} + L * MMD(D_L(Laplacian(G(z)), D_L(Laplacian(x))))+$$
$$+S * MMD(D_s(Shapes(G(z)), D_s(Shapes(x))))$$

Where D_L and D_S are 3 layers MLP with batch normalization and Relu activation function after each of the first 2 layers. Both networks are trained togethor with the learned kernels. D_L trained to maximize $MMD(D_L(Laplacian(G(z)), D_L(Laplacian(x))))$, and D_S trained to maximize MMD(D$_s(Shapes(G(z)), D_s(Shapes(x))))$

GCN type	GCN count	GCN	GCN + L	GCN + L + S
Constant	1	5.83%	12.45%	10.55%
Constant	3	6.29%	9.27%	6.21%
Replacing	1	6.46%	8.17%	10.65%
Learned	1	13.42%	16.13%	11.371%
all	1 from each	7.69%	5.01%	14.15%

Table 1: Peak valid unique score with unlearned Laplacian(L) and shapes(S) kernels.

4 Experimental results

4.1 Valid unique score

We tested the models on the valid unique score. The number of unique and valid molecular created per 100 generated graphs. The score does count duplicate graphs, but only once. The reason we used this method instead of the valid score tested used in the MolGAN paper, is that a generator which create the same graph all the time would get a perfect valid score. This option is not just theoretical, and it did happened to us during one of the training. The score were computed on group of 1024 graphs on every 10 iterations, and we used the peak score over every 50 onsecutive computed iterations.

4.2 Results

We trained the models using multiple types of GCN kernels (see GCN kernel) (table 1), and tried different number of kernels. The The learned kernel together with the Laplacian kernel performed the best with a valid unique score of 16.13%. The shapes kernel did help for some configurations, but sometimes hurted the model. Training multiple GCN kernels didn't seem to help the results, besides when we trained 1 of each GCN kernels together with the Laplacian and the shapes kernel, which was our best result using the untrained shapes kernel.

4.2.1 Training the Laplacian and Shapes kernel

We also wanted to test the effect of training the Laplacian and Shapes kernels as we described at 3.2.3. We were able to improve the valid unique score by 3.39% when using both the Laplacian kernel and the Shapes kernel. While using both kernels resulted with the best valid unique score, the graph quality of the Shapes kernel was the highest (figures 2,3).

4.2.2 The N_{critic} hyper-parameter

One important thing to choose when training the different models is the N_{critic} hyper-parameter, which counts for the number of training iterations of the learned kernels for each training iteration of the generator. Choosing a low value might lead to the network not performing the best it could, and choosing a high value might make the generator training to be useless. The N_{critic} value was different for different setups. When we trained the generator only on the GCN kernel, we set N_{critic} hyper-parameter to 3. When we trained the

GCN type	GCN+L	GCN+S	GCN+L+S
Constant	11.367%	7.9%	11.436%
Replacing	9.434%	7.96%	10.87%
Learned	17.92%	14.80%	19.52%

Table 2: Peak valid unique score with Learned Laplacian(L) and/or Shapes(S) kernel.

Reward	Laplacian kernel	Score
No	No	12.29%
Yes	No	13.23%
Yes	Yes	23.26%

Table 3: MolGAN'S peak valid unique score.

generator with the unlearned Laplacian kernel, the generator got an edge over the learned kernel, and we set the N_{critic} hyper-parameter to 20 to get the best results. On the other hand, when we used the learned Laplacian kernel, we had to set the N_{critic} hyper-parameter to 1. This showed us that the Laplacian kernel was very helpful for the generator learning.

4.2.3 Comparing the result to the MolGAN approach

We tested the peak valid unique scores of the MolGAN approach(table 3) and compared the results to ours. The MolGAN network performed worst then our best network even when it used it's reward system. We wanted to test the effect of the unlearned Laplacian kernel on the MolGAN network, so we added $L * MMD(Laplacian(G(z), Laplacian(x)))$ to the Molgan's generator loss function. This improved the MolGAN performance by more then 10%.

5 Discussion

We found in this project that using kernels in graphs can be very useful when trying to distinguish between graphs, and they can contribute when they are trained together with a GCN. The Laplacian kernel proved to be effective because using him resulted with a higher valid unique score and because when we used it only on the generator loss we could train the generator much less then the learned kernel. The Laplacian kernel also helped us to improve the MolGAN valid unique score which shows that it can be used with models which doesn't use MMD loss for other parts of their loss functions. The learned Shapes kernel was also very useful, and helped us to improve the molecules quality. While both kernels did improve our training, we do believe they can even improve more graphs with no bond types, because they doesn't distinguish between bond types.

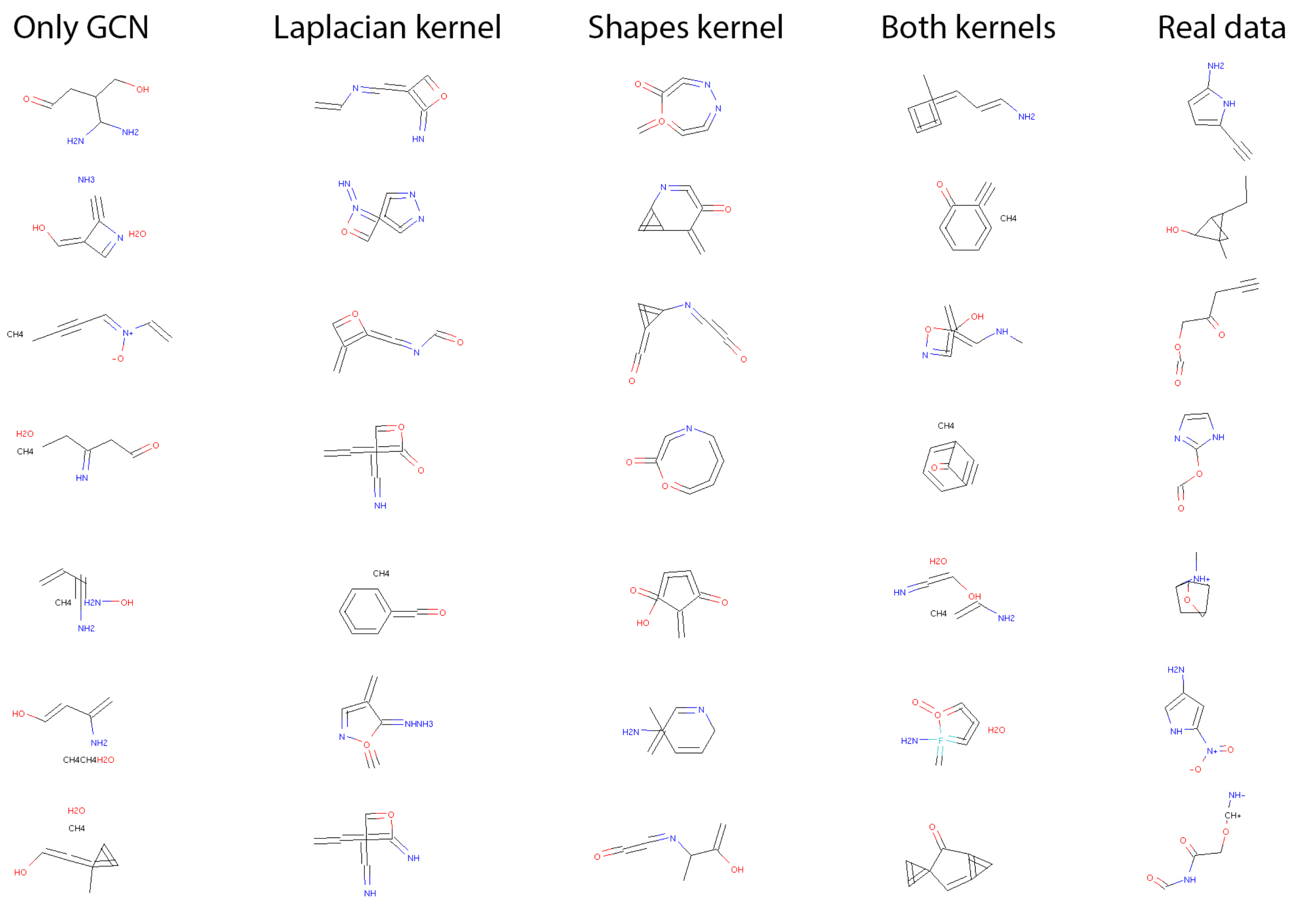

Figure 2: Handpicked molecules created with different learned kernels

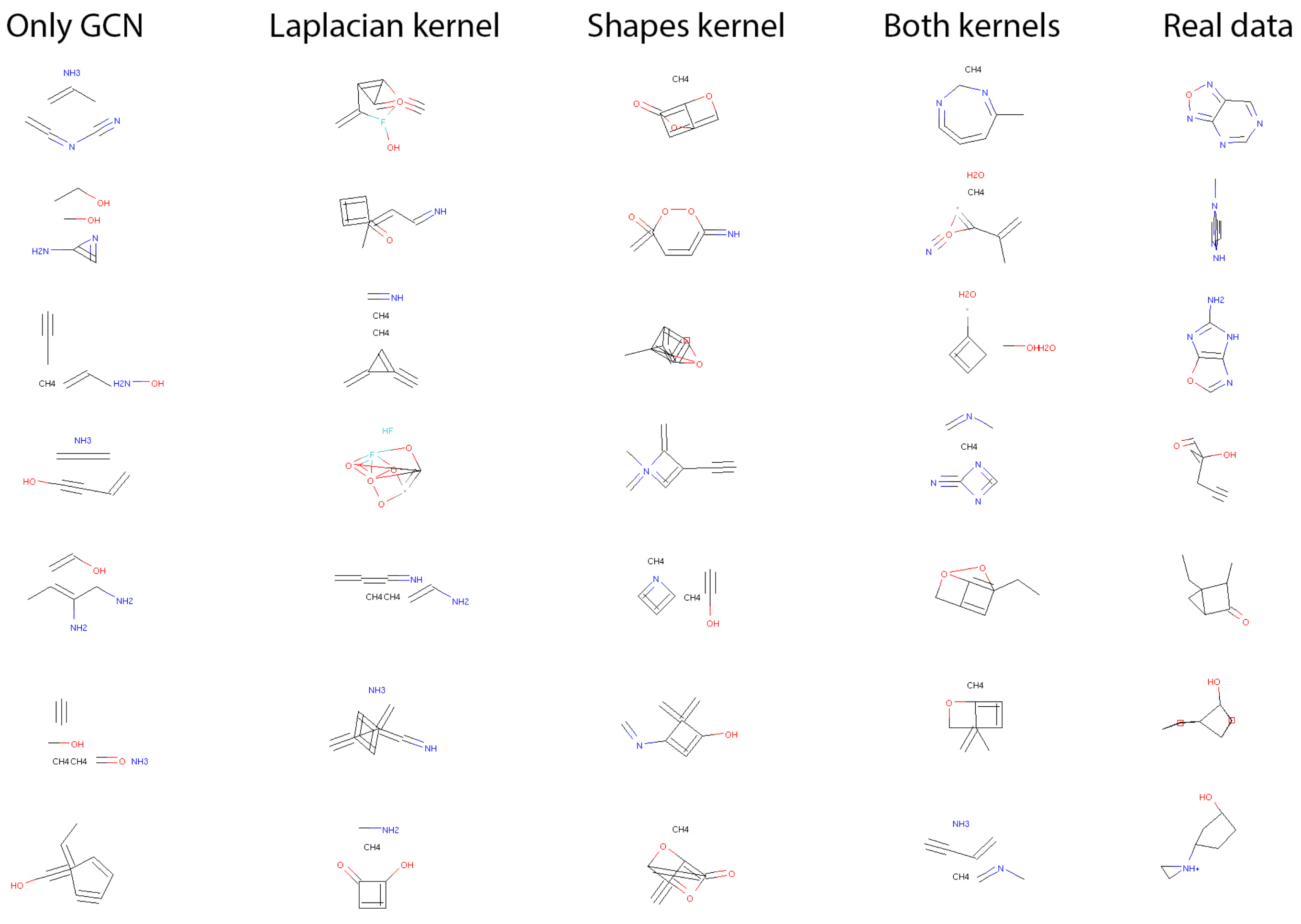

Figure 3: Random molecules created with different learned kernels

6 Code

You can find the code in the following GitHub repository:
https://github.com/eran88/graph-kernel
You can view the instructions for this article's hyper parameters configuration here
To recreate the experiments described in this article view this instructions

Code attribute: We used the MolGAN pytorch repository as a base for our code. We also this MMD loss function and MMD loss function with multiple bandwidth parameters.

References

[1] Bidisha Samanta, Abir De, Gourhari Jana, Vicenç Gomez, Pratim Kumar Chattaraj, Niloy Ganguly, Manuel Gomez-Rodriguez. "NEVAE: A Deep Generative Model for Molecular Graphs". In: (2020).

[2] Chence Shi, Minkai Xu, Zhaocheng Zhu, Weinan Zhang, Ming Zhang, Jian Tang. "GRAPHAF: A flow-based autoregressive modelfor molecular graph generation". In: (2020).

[3] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton,Jan Eric Lenssen, Gaurav Rattan, Martin Grohe. "Weisfeiler andLeman Go Neural: Higher-order Graph Neural Networks". In: (2019).

[4] Chun-Liang Li, Wei-Cheng Chang, Yu Cheng, Yiming Yang, Barnabás Póczos. "MMD GAN: Towards Deeper Understanding of Moment Matching Network". In: (2017).

[5] Eric Jang, Shixiang Gu, Ben Poole. "Categorical Reparameterization With Gumbel-Softmax". In: (2017).

[6] Gabriel Guimaraes,† Benjamin Sanchez-Lengeling, Carlos Outeiral, Pedro Luis, Cunha Farias, Alán Aspuru-Guzik. "Objective-Reinforced Generative Adversarial Networks (ORGAN) for Sequence Generation Models". In: (2018).

[7] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair†, Aaron Courville, Yoshua Bengio. "Generative Adversarial Net". In: (2014).

[8] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin,Aaron Courville. "Improved Training of Wasserstein GANs". In: (2017).

[9] Martin Arjovsky, Soumith Chintala, Leon Bottou. "Wasserstein Generative Adversarial Networks". In: (2017).

[10] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, Max Welling. "Modeling Relational Data with Graph Convolutional Networks". In: (2017).

[11] Nicola De Cao, Thomas Kip. "MolGAN: An implicit generative model for small molecular graphs". In: (2018).

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. “Playing Atari with Dee pReinforcement Learning”. In: (2013).

[13] Yibo Li, Liangren Zhang, Zhenming Liu. “Multi-Objective De Novo Drug Design with Conditional Graph Generative Model”. In: (2018).

[14] Yujia Li, Kevin Swersky, Richard Zemel. “Generative Moment Matching Networks”. In: (2015).