Applying Deep Learning Methods on Detecting Cycles on Graphs

By

Ka Hei Michael Chu

SENIOR THESIS

Submitted in partial satisfaction of the requirements for Highest Honors for the degree of

BACHELOR OF SCIENCE

in

MATHEMATICAL ANALYTICS AND OPERATIONS RESEARCH

in the

COLLEGE OF LETTERS AND SCIENCE

of the

UNIVERSITY OF CALIFORNIA,

DAVIS

Approved:

Jesús A. De Loera

June 2020

ABSTRACT. This thesis is about using deep learning to identify whether a n-vertex simple undirected graph has a k-length cycle. I present the details of my work with Professor Jesús A. De Loera on using artificial neural network architectures to perform such classification problem.

It is found that the eigenvalues alone are a very telling factor of whether a graph has a k-cycle, as using only eigenvalues as features for the neural network achieves accuracy close to providing full information of the graph into the neural network, despite the abstraction of information. This indicates that the eigenvalues contain important information about the graph's topology.

Another surprising finding is that recurrent neural network, typically used for temporal or otherwise continuous data such as speech and video, outperforms feed-forward neural network on the task of cycle-detection, which is discrete and combinatorial.

Contents

CHAPTER 1

Background

Artificial intelligence, and neural network in particular, has been a hot topic recently in Computer Science. One might have heard of the successful results of artificial intellegence in complicated tasks such as synthesized speech, computer vision, and beating Chess and Go masters in their own games. In this thesis, we apply neural networks to solve a mathematical problem. This chapter provide an intuitive introduction to the concept of artificial neural network.

1.1. Machine learning

Machine learning is the study of computer algorithms that improve through experience without being explicitly programmed. In conventional computer programming, a machine is instructed to perform a predefined process. In contrast, for a machine learning paradigm, the machine attempts to perform tasks without being explicitly programmed to do so.

For example, in conventional programming, we instruct the machine to run the process "return $y = 2x + 1$" and provide the machine with input data $x = [0, 1, 3, 5, 6, 9]$, the machine would return the output data $y = [1, 3, 7, 11, 13, 19]$. In machine learning, we feed the machine with input $x = [0, 1, 3, 5, 6, 9]$ and desired output $y = [1, 3, 7, 11, 13, 19]$. The machine would attempt to learn the underlying mathematical relationship between the x and y. After the machine 'learns' the linear relationship between the input and output data, perhaps we would want to know how much is $2 \times 4 + 1$. We provide the machine with additional input $x' = 4$ and, hopefully, the machine would return $y' = 9$ — or some values numerically close to it.

Machine learning excels in performing tasks that are difficult to hand-craft an algorithm for, as the machine discovers latent relationship or structures within the data. Common applications of machine learning are computer vision and data-mining.

Machine learning can be categorized into three subtypes:

1.1.1. Supervised learning.

The machine, provided with input data and corresponding output, is tasked to learn the mathematical relationship between the input data and the corresponding output. The case of 'learning' $y = 2x + 1$ above is an example of supervised learning. Supervised learning can be further demarcated into two sub-categories:

Definition 1. In a **classification**, the set of possible outputs is finite. An example of classification problem would be the set of input data $x = [0, 1, 3, 5, 6, 9]$ and outputs $y = [0, 1, 1, 1, 0, 1]$, with the underlying relationship

$$y = \begin{cases} 1 : & x \text{ is an odd number} \\ 0 : & \text{otherwise} \end{cases} .$$

The output in classification is also called **label**.

Definition 2. In a **regression**, the set of output is not necessarily finite or even countable. For instance, the example of $y = 2x + 1$ above is a regression, where the output range is the set of Real numbers.

1.1.2. Unsupervised learning. The machine attempts to find underlying patterns or structures in a given data set. Unlike Supervised Learning, the machine in unsupervised learning is not given outputs or labels, hence the learning is 'unsupervised' as there are no labels to verify whether the machine-predicted output is right or wrong.

1.1.3. Reinforcement learning. The machine is put into a dynamic learning environment where the machine receives rewards based on its actions, and its objective is to maximize cumulative rewards. In contrast, the learning environment of both supervised and unsupervised learning are static,as the environment is the data set itself. Reinforcement learning is commonly used in action-based tasks, such as building an AI for chess or for video games.

In this thesis, we use machine learning, and in particular **artificial neural networks**, to learn to solve a classification problem that would otherwise take an exponentially long time to perform in classical programming.

1.2. Artificial neural networks

An **Artificial neural network** (NN) is a machine learning method loosely related to the biologicial structure of neural networks in an animal brain. A NN is a collection of **layers** of interconnected **artificial neurons**, or nodes, that receive and transmit signals to other neurons. Data is passed into the neural network, in which the neurons transform the data and condense the transformed data signals into a machine-predicted output.

Deep learning is the use of NNs with multiple hidden layers for machine learning. There was a distinction between artificial neural networks and deep learning, as the latter excludes single hidden-layer networks from the general notion of NN. However, given the growth of computational power, deep learning has become not only feasible but also very approachable - a multiple layer network with width counted in hundreds can be trained

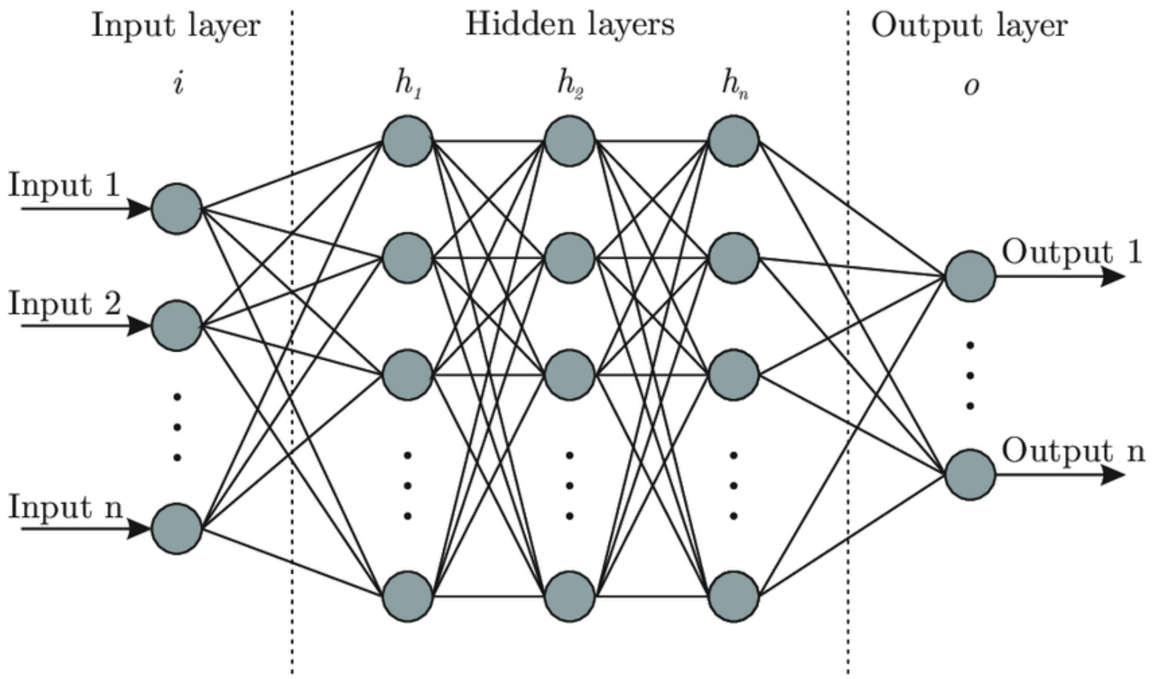


Figure 1. Visualization of the architecture of an artificial neuron network. Image taken from [**2**].

on a laptop. Now, the two terms are used almost interchangeably, as most researches and applications of NNs are multi-layered.

Figure 1 is a visualization of an artificial neural network. Each node represents a neuron, and each edge between neurons (directed from left to right) represents a neural connection. Note that this is a feed-forward neural network, a subtype of NN to be discussed in the coming section.

The performance of the NN is evaluated by a **loss function**, which compares all labels and machine-predicted output to produce a **loss value** quantifying the machine prediction's deviation from the labels. Weights of neuron connections (and their output signal thereof) are adjusted according to the loss value to minimize the loss.

The mathematical definitions of these terms are detailed below.

1.2.1. Neurons. Artificial neurons are the atomic components of a NN. An artificial neuron receives input signals from other neurons, transform the signals into a numerical output value, and sends the output value to other neurons. Generally, a neuron takes the weighted sum of all input signals plus the bias value of the layer, transform the sum by the **activation function** of the neuron, and output the transformed sum to other neurons.

To give an abstract mathematical notion, the output value of a neuron is

$$c(w \cdot a + b)$$

where $c(x)$ is the activation function of the neuron, a is the vector of input signals from other nodes, w is the corresponding connection weights corresponding to the inflowing neuron signals, and b is the bias value for the layer.

Some desirable properties for an activation function include:

- Nonlinearity
- Choice of range: finite range makes training more stable, but unbounded range makes training more efficient
- Continuously differentiable
- Monotonocity
- Smooth functions with smooth derivatives
- Approximates identity near origin

Common choices of activation are listed below:

Definition 3. The **Rectified linear unit** (ReLU) is a piecewise linear function defined as:

$$ReLU(x) = max(0, x)$$

Composition of ReLUs can represent any piecewise-linear function. ReLU has codomain $[0, \infty)$. The function is continuous, but not differentiable at $x = 0$. Nevertheless, ReLU works well in practice.

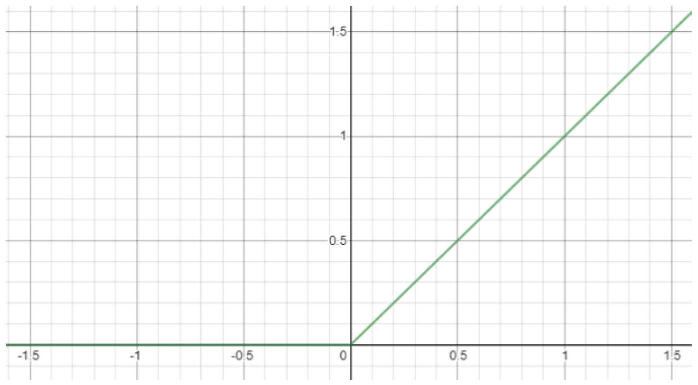


Figure 2. graph of ReLU.

Definition 4. **Sigmoid**, also known as the logistic function, is a smooth function defined as:

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

A closely related activation function is **hyperbolic tangent**:

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

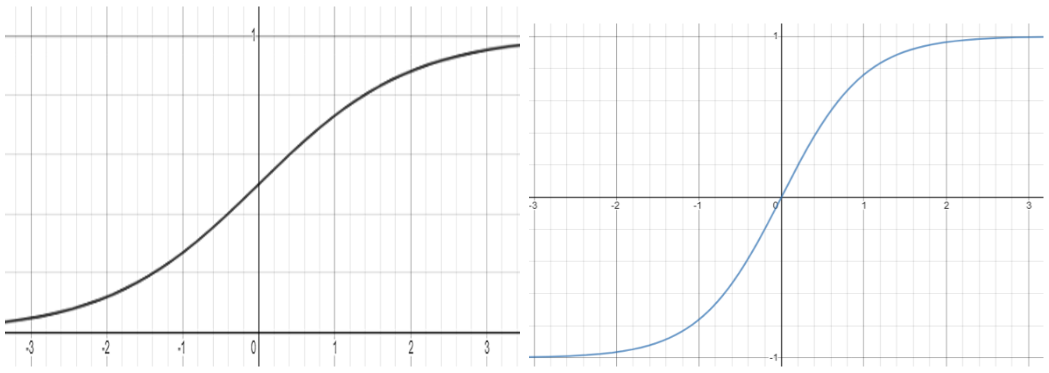


FIGURE 3. Graph of Sigmoid (left) and *tanh* (right). Note that sigmoid is centered at $(0, 0.5)$ while *tanh* is centered at the origin. The two functions are equivalent up to rigid transformation.

Note that $\tanh(x) = 2\sigma(2x) - 1$. Since sigmoid has range $(0, 1)$ and hyperbolic tangent has range $(-1, 1)$, one may opt for one or another depending on the desired range of neuron output signals.

DEFINITION 5. **Softmax** normalizes an input vector into a probability distribution. Softmax takes an input of vector $z = (z_1, \ldots, z_n) \in \mathbb{R}^n$:

$$softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \qquad (i = 1, \ldots, n)$$

Softmax normalizes each component z_i of the vector to the range of $(0, 1)$, and the normalized vector have unit sum. Thus the output can be treated as a probability distribution, where larger input component corresponds to a larger probability value.

Note that unlike the previous activation functions, where the input is the weighted sum of inflowing signals plus bias, softmax's input is the vector of signals.

1.2.2. Layers. Neurons in a NN are organized into a collection of **layers** to control the flow of signals. The flow of information starts from the **input layer**, where each neuron represents a datum from the data set. Note that the input layer neurons do not have activation functions. The input layer passes the data to the **hidden layers**, where learning and computation takes place. Each hidden layer propagates its output signals to its successive hidden layer, and finally to the **output layer**, where the neurons compute the output of the NN model. The loss value of the output is calculated, and the weights and biases in the network are adjusted using gradient descent.

The **width** of a layer is the number of neurons on the layer. The computational and learning capacity of a neural network depends not only on the number of neurons in the network (or the sum of widths of hidden layers), but also the number of hidden layers.

Definition 6. A layer is **dense** if each neuron in the said layer sends its output signal to every neuron in its successive hidden layer.

1.2.3. Loss Function. The **loss function**, or **cost function**, of a NN is a quantification of how much the NN's classification result deviates from the actual result, i.e. the labels. The machine then recalibrates the weights of neural connections and the bias value of layers. Listed below are some common choices of cost functions.

Definition 7. The **quadratic cost function** is defined as

$$C(w,b) = \frac{1}{n}\sum_x \|y(x) - a(x,w,b)\|^2$$

where

- n is the size of training data set;
- x is the data in training set;
- $y(x)$ is the output or labelled value corresponding to the data point x;
- w is the entire set of connection weights in the NN;
- b is the vector of biases in all the layers in the NN;
- $a(x,w,b)$ is the NN's prediction of x's label or regression output, given the specifications of w, b

which is essentially the sum of squared prediction errors. This cost function is a common choice for regression modeling.

Definition 8. **Cross-entropy** is a loss function for binary classification, defined as:

$$CE = -\frac{1}{n}\sum_x [y(x)\ln(a(x,w,b)) + (1-y(x))\ln(1-a(x,w,b))]$$

with the same notations of a, x, w, b as above. Here, y is the binary classifier variable (so it is either 0 or 1), and the activation function a of the output layer generally has output range $(0,1)$. For instance, sigmoid is a suitable choice for a.

1.3. Types of neural networks

1.3.1. Feed-forward neural networks. In a **feed-forward neural network** (FNN), information flows in one direction. A FNN is composed of sequenced dense hidden layer(s), and data is propagated from the input layer to each successive hidden layer, and finally to the output layer. Note that in a FNN, neurons do not connect to other neurons from the same layer. Thus, if we consider the NN as a graph with neurons as vertices and information flow as directed edges, then a FNN forms an acyclic graph. If we ignore the direction of information flow, then graph of FNN forms a L-partite graph, where L is the number of layers.

The weights in a FNN can be described more succinctly using matrices. Let L_n denote the n^{th} layer and $|L_n|$ denote its width. Since each neuron in the n^{th} layer is connected to every neuron in the $(n+1)^{\text{th}}$ layer, the connection weights between these two layers forms a matrix $W^{(n)} \in \mathbb{R}^{|L_n| \times |L_{n+1}|}$, where $W_{ij}^{(n)}$ is the weight of the output signal from the i^{th} neuron of L_n to the j^{th} neuron of L_{n+1}.

The illustration in figure 1 is in fact a feed-forward neural network with hidden dense layers. In contrast, some alternative neural network architecture allows data signals to be sent to previous layers or to the same layer.

1.3.2. Recurrent neural networks. Recurrent Neural Network (RNN) is a type of NN that is typically used for temporal or otherwise contiguous data. The distinguishing feature of an RNN is that in each training set, the NN remembers data from the previous training steps. Common tasks for RNN are audio, speech, video and handwriting recognition.

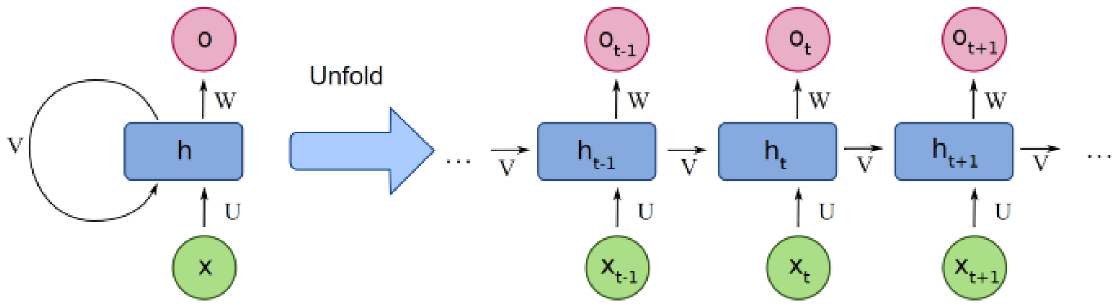


FIGURE 4. A high-level visualization of the RNN architecture. [**10**].

An RNN inherits the architecture of a FNN, except neurons also send their output signals back to themselves. Figure 4 is a visualization of a RNN. The green node x_t the t^{th} datum, h_t is the state (weights, biases and activation functions) of the hidden layers at the t^{th} training step, and σ_t is the output of the NN at the same training step. After training step t, the neuron signals in step t from the hidden layers are passed back to the hidden layers for step $t+1$.

A NN could be a hybrid of FNN and RNN by having some layers being dense and some being recurrent. For example, one could have a NN with two layers of LSTM (a type of recurrent layer) followed by a dense feed-forward layer. Since RNN can be described as an iteration of the FNN, we would still put 'hybrid' networks under the umbrella of the RNN class.

1.4. Underfitting, overfitting and regularization

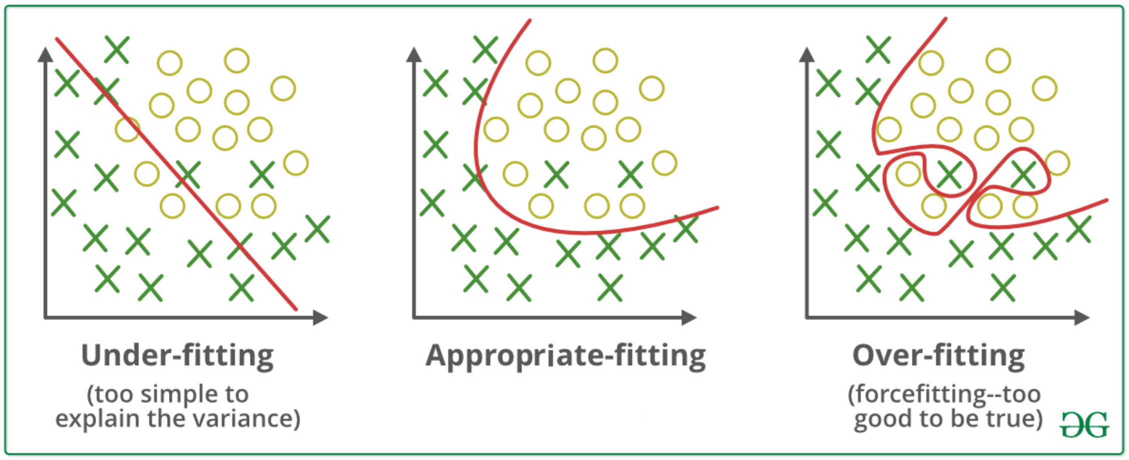

Under-fitting
(too simple to explain the variance)
Appropriate-fitting
Over-fitting
(forcefitting--too good to be true)

FIGURE 5. A depiction of a machine trying to separate the crosses crom the circles. The example is in fact an instance of support vector machine, an unsupervised machine learning method. Image taken from [**6**].

Figure 5 provides a visual example of underfitting and overfitting. Here, the machine is tasked with drawing a curve to separate out the green crosses from the yellow circles on a $2D$ plane. The curve on the left is underfitted, as the curve is too simplistic and does not fully capture the latent structure of the point clouds. The curve in the middle is a better fit for the data. The curve on the right, albeit being a better fit for the data, is considered an overfit, as the curve is tailored to the specific data set. In reality, data could be noisy, and the few crosses spilled over to the circle point cloud could be due to noise. If the curves are fitted into another data set with the same latent structure or distribution, then the rightmost curve would perform signifcantly worse than the middle curve.

The main purpose of machine learning is to have the machine learn the underlying patterns or structure in a given set of data. The machine is expected to generalize and therefore be able to predict or regress on new data. But the machine could learn to memorize the data - without actually 'learning' anything substantial — to perform well on the given data set. To test whether the machine is learning but not memorizing, we test the machine's performance on a data set different from the one that the machine is trained on.

Therefore, we split the data set into training set and testing set. The machine is trained on a training data set, and once the machine is sufficiently trained and performs well on the training set, the model is tested on the separate testing set to evaluate its true performance. The machine's training performance improves as it learns more from the data, but eventually the improvement plateaus. Initially, the machine picks up on general patterns within the data, as they provide the greatest improvement on loss. As most patterns within the data are picked up, the machine struggles to make improvement and resorts to memorizing the input data.

The problem of **underfitting** is when the machine has not learned all that it could from the data set. The solution is to simply increase the number of epochs so that the machine revisits the same data set for multiple times. If a few extra epochs give the model a significant increase in performance, then the model is likely to be underfitted before the increase of epochs. It would then be tempting to conclude that more epochs would always benefit the model - this is not the case, as we would see below.

The other problem is **overfitting**, when the machine performs well on the training set but not on the test set. This means that the machine might be memorizing the data instead of generalizing it, resulting in a rapid performance drop on the test set. The natural solution to the problem is to either decrease the number of epochs, or introduce more training data. However, data is not always easy to come by, so we don't take the size of data as a hyperparameter that can be increased at will. Since we would underfit if we have "too few" epochs, and would overfit if we have "too many" epochs, this implies an optimal amount of epochs - and therefore a cap in performance — for the model training. The choice of epochs is often a matter of trial-and-error, but one can also instruct the machine to stop the training process once it fails to improve the loss function significantly.

Figure 6 depicts a typical progression of performance of a NN in its training process. The horizontal axis is the number of epochs, and the vertical axis is the loss of the model. The model's performance in the training set (black curve) improves and the loss decreases as number of epochs increases, but the magnitude of improvement is also decaying. The model always perform better in the training set with more repeated training on the same data set. Performance on test set (red curve) also improves as epochs initially increases, but eventually the improvement flats out. The model even started doing worse on the test set as it is trained excessively, which is a sign of overfitting.

Since there is a trade-off between underfitting and overfitting, methods to surpress overfitting (without limiting the number of epochs) would increase the performance of the machine. Listed below are some methods of dealing with overfitting.

1.4.1. Downscaling neural network model. While having a larger neuron network (i.e. more neurons in the network) has more computational power and learning capacity, it

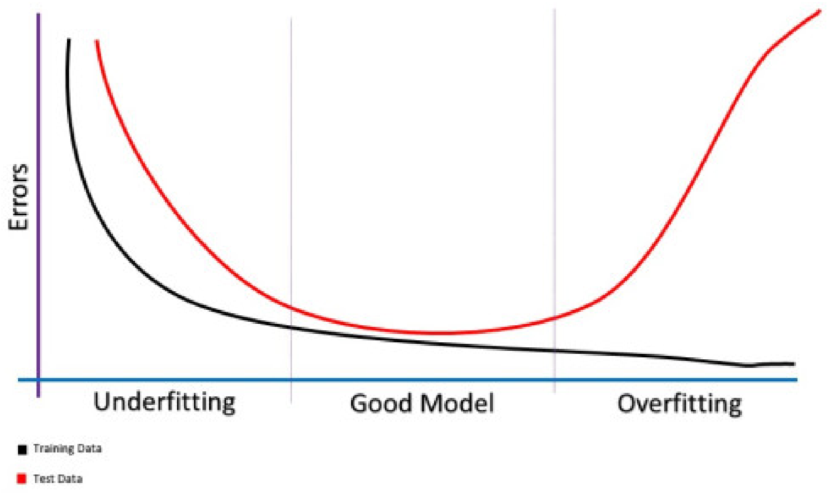


FIGURE 6. A graph of model accuracy on training and testing set over the number of epochs. Image taken from [**4**].

also has a higher capacity to memorize data. An excessively large NN may not perform significantly better than a 'right-sized' network, but the larger model is more computationally expensive, and leads to faster overfitting. A scientific rule-of-thumb is to favor a simpler model or hypothesis over a more complicated one if both explains a phenomenon equally well, so downscaling the NN to prevent overfitting is both practically and philosophically sound.

1.4.2. Dropout. In each training step, each neuron in the layer with dropout rate $p \in [0, 1]$ has a probability $1-p$ of not activating — its output signal is effectively multiplied by 0. The random dropout of neurons prevents the NN from over-relying on a few specific nodes, as the nodes have a non-trivial chance of being de-activated. The dropout also mimics an ensemble of networks - since each neuron in a layer of width n has a chance of being 'turned off' in each learning step, the training process is as if 2^n are being trained all at once (albeit sharing some weights) with the computational cost of training only one NN. The choice of nodes to be dropped out is computationally inexpensive, and dropout improves NN's training speed in real time as the capacity of the network is randomly reduced. [**12**]

1.4.3. L1 regularization. $L1$ regularization, also known as **LASSO** regularization, adds a penalty term $\alpha\|w\|_1$ proportionate to the $1-$norm of NN weights to the loss function.

Under the modified loss function, the machine takes into account not only the deviation of model predictions from the real output, but also the total weight of the neural network. The NN is therefore penaltized for 'learnings', such as memorization, that do not significantly improve the NN's performance.

The choice of penalty rate α is a matter of heuristics and there is no notion of a mathematically optimal α.

1.4.4. L2 regularization. $L2$ (or L^2) regularization, also called **ridge** regularization, adds a penalty term $\alpha\|w\|_2$ proportionate to the $2-$norm of weights to the loss function. The concept is similar to that of $L1$, but $L2$ penaltizes the NN based on the square of weights. Large weights are penaltized heavier than it would be under $L1$, so $L2$ is here mainly to combat the network's over-reliance on a few singular nodes and therefore enforce the learning to be done 'evenly' across all the neurons. Like $L1$, the penalty rate for $L2$ is a matter of trial and error.

CHAPTER 2

Graphs

This chapter provides some intermediate mathematical background on graph theory, as well as to provide some motivation on the use of machine learning to tackle a graph theory problem.

2.1. Adjacency matrices

Let $G = (V, E)$ be a simple graph with nodes $1, \dots, n$. The **adjacency matrix** of G is a matrix $A \in \mathbb{R}^{n\times n}$ where $A_{i,j} = 1$ if $(i, j) \in E$ and $A_{i,j} = 0$ otherwise. Given an adjacency matrix of G, we can check whether vertices i can reach j in exactly k steps by computing $A^{k-1}_{i,j}$. Furthermore, if G is undirected, A is symmetric, so A's eigenvalues are all real, and the eigenvectors form an orthonormal basis. [**1**]

Adjacency matrix is a representation that provides full information on the graph. It also uniquely identifies a graph up to vertex isomorphism. Hence, adjacency matrix of a graph and the graph itself are often referred to interchangeably. In particular, we say that the (adjacency matrices of) graphs have eigenvalues.

Eigenvalues of a graph, even if they do not uniquely identify the graph (see co-spectral graphs), provide useful latent information about the graph. For example, the largest eigenvalue is related to the maximum degree and average degree of the graph:

PROPOSITION 1. *For a graph G with eigenvalues λ_i and each vertex having degree d_i:*

$$\max\{\bar{d}, \sqrt{d_{max}}\} \le \lambda_{max} \le d_{max}.$$

The whole set of eigenvalues are shown to be useful for supervised learning [**11**].

2.1.1. Erdős–Rényi model. Erdős–Rényi model is a probabilitistic scheme for generating random simple, undirected graphs. For a specified probability $p \in [0, 1]$, the scheme $G(n, p)$ generates a graph with n vertices where each edge has an activation probability p. The scheme generates graph that have a lower degree of clustering, compared to 'real-life' social networks.

The activation probability p is closely related to the connectedness of the graph:

PROPOSITION 2. *If $p < \frac{(1-\epsilon)\ln n}{n}$, then a graph generated by the $G(n, p)$ scheme is very likely to have isolated vertices. If $p > \frac{(1+\epsilon)\ln n}{n}$, then a graph from the $G(n, p)$ scheme is* *very likely to be connected. Therefore $\frac{\ln n}{n}$ is a threshold for the connectedness of $G(n,p)$.* [**8**]

An alternative formation for the scheme is: for a specified $m \in \mathbb{N}, G(n,m)$ generates a n-graph with a random m-combination from the edge set. Both the chanced edge-activation formulation and the fixed-number-of-edges formulation give effective control on the density of the generated graph.

We use the $G(n,p)$ scheme to generate one of the dataset in our experiments.

2.2. Non-Deterministic algorithms for finding cycles

There are a lot of known common algorithms, such as breadth-first search, that exhaustively (and, therefore, deterministically) finds cycles within a graph. Although the problem of finding certain types of cycles could be done in less than exponential time, the general problem of finding a k-cycle in a graph is regarded to be NP-hard. Therefore, we look into non-deterministic algorithms to perform such tasks in a much quicker way. [**13**]

Suppose we want to find a k-cycle in a graph $G = (V,E)$. Define a random coloring scheme $c : V \to [k]$ where each node is assigned a color uniform-randomly from the color set $[k] = \{1, 2, \dots, k\}$. Now, suppose that there is a k-cycle $v_1 \to v_2 \to \cdots \to v_k \to v_1, \quad v_i \in [k]$. The probability $\mathbb{P}(c(v_i) = i)$ is $\frac{1}{k}$ by uniform-randomness, so the probability $\mathbb{P}(c(v_i) = i | i \in [k]) = k^{-k}$.

We say that the path $v_1 \to \cdots \to v_k$ is **well colored** if $c(v_i) = i \quad \forall i \in [k]$. Likewise, a cycle is well-colored if it contains a well-colored path.

If we repeat the random coloring $100k^k \log(|V|)$ times, the probability that the cycle C is not well colored in any of the colorings is

$$(1 - \frac{1}{k^k})^{100k^k \log(|V|)} \leq (\frac{1}{e})^{100 \log(|V|)} = |V|^{-100}.$$

Therefore, the probability that C is well-colored in some of the colorings is $1 - |V|^{-100}$.

For a fixed coloring $c(v)$, let $G' = (V, E')$ be a directed graph from $G = (V,E)$, where $u \to v \in E'$ iff $(u,v) \in E$ and $c(v) = c(u) + 1$.

Proposition 3. *Given $c(u) = 1, c(v) = k$, u can reach v in G' iff there is a well-colored path $u = x_1 \to x_2 \to \cdots \to x_k = v$ in G.*

Proof. ($\Longleftarrow$) Suppose there is a k-path $u = x_1 \to x_2 \to \cdots \to x_k = v$ in G where $c(x_i) = i \quad \forall i \in [k]$. Then $x_i \to x_{i+1} \in G'$, so u can reach v in G'.

($\Longrightarrow$) Suppose $u = x_1 \to \cdots \to x_\ell = v$ in G' for some $\ell \in \mathbb{N}$. Since $x_i, x_{i+1} \in G'$ iff $x_i, x_{i+1} \in G$ and $c(x_{i+1}) = c(x_i) + 1$, we have that the colors on the path increases by 1 on every step. Since $c(u) = 1$ and $c(v) = k$, ℓ is necessarily equal to k, so $u = x_1 \to x_2 \to \cdots \to x_k = v$ is well-colored. $\square$

Similarly, if we have a well-colored cycle $x_1 \to \cdots \to x_k \to x_1$ in G, then x_1 can reach x_k in G'. Conversely, if we have $u, v \in V$ where $c(u) = 1, c(v) = k$ and there is a path from u to v in G', then there is a well-colored cycle in G containing the said $u-v$ path.

To find a well-colored k-cycle, we first find all pairs of vertices $u, v \in G'$ where u can reach v. For each pair (u, v), we then check if $v \to u$ in G, $c(u) = 1$, and $c(v) = k$. The check can be done in $O(|E|)$ time.

To find all the $u - v$ pair, we can use a breadth-first search, which runs in $O(|E| \cdot |V|)$ time. Another way is to use make use of adjacency matrices - construct the adjacency matrix A for G', then compute A^{k-1}. $A^{k-1}_{u,v} \neq 0$ iff u can reach v in G' in k-steps.

For an algorithm of $O(k^k)$ such as this one, we can find a cycle of $O(\frac{\log(|V|)}{\log(\log(|V|))})$ in polynomial time.

2.3. Contemporary works on graph theory

Graph theory is an important analysis tool in Mathematics and Computer Science, and a handy abstraction of data or systems in different forms. We discuss below some contemporary open problems on graph theory as a motivation to our experiments.

In 1995, mathematicians Paul Erdős and András Gyárfás stated the Erdős–Gyárfás conjecture: every graph with minimum degree 3 contains a simple cycle whose length is a power of two.

By an extensive computer search, Gordon Royle and Klas Markström found that any counter-example must have at least 17 vertices. Markström found some near counter-examples (see figure 1): he found four 24-vertices graphs that only have one $16-$cycle. [**7**]

A special case of the conjecture has been proven. In 2013, Christopher Carl Heckman and Roi Krakovski proved the conjecture for the case of 3-connected cubic planar graphs.[**5**] This conjecture is, in fact, the original motivation to this thesis, as we attempt to use 'quick' ways to sift through a large number of graphs and dig for a counter-example.

Another interesting conjecture was stated by Thomassen, inspired by the observation that bipartite graph has no even-length cycles. The conjecture states that: every graph G whose minimum degree is larger than k contains a cycle of length $(2s \mod k)$ for natural numbers s, k. [**9**] Dean further the conjecture: a graph with minimum degree no less than $k \geq 3$ has a cycle whose length is a multiple of k. Dean showed that these two conjectures are true for special cases $k = 3, 4$. [**3**]

The fact that a great deal of the work done on the conjecture involves some degree of brute-force by computer motivates the usage of other methods reliant on computing power - such as artificial neural network.

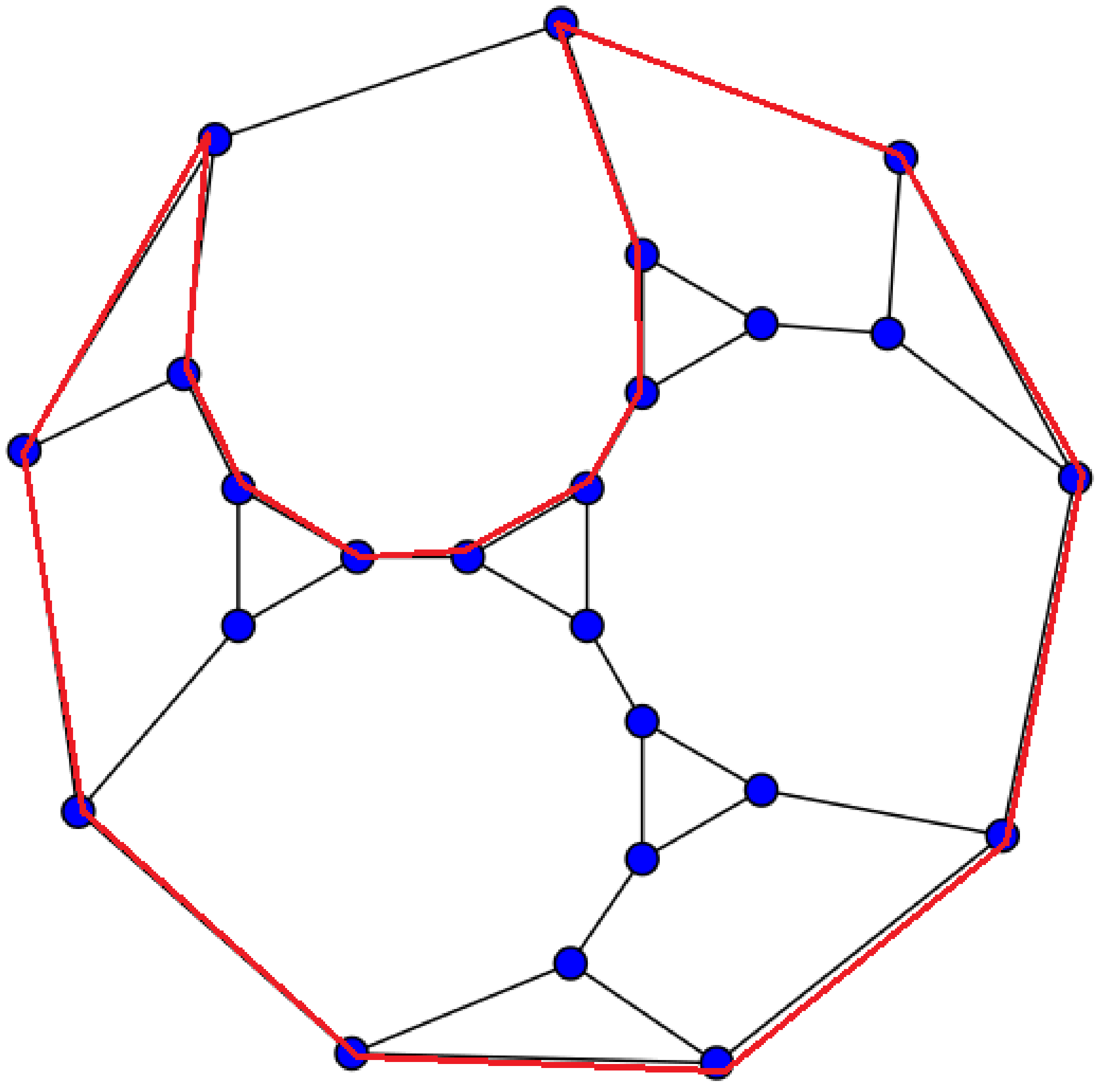


FIGURE 1. One of the near-counterexample found by Markström. The only 2^n cycle is the 16-cycle marked red on the graph. [**7**]

CHAPTER 3

Experiments

3.1. Datasets

We use three different sets of graphs to train the NNs on. The first set is all (12346) the 8-vertices graphs, up to isomorphism. The second one is a random sample (27466) of all (274663) 9-vertices graphs, up to isomorphism. The third one is a collection (140000) of 20-vertices graphs randomly generated from the Erdős–Rényi model. For activation probabilities $p = 0.07, 0.08 \ldots, 0.13$, we generated ten thousand graphs with 8-cycle(s) and ten thousand graphs without 8-cycles. We aim to produce graphs that are connected but not dense, so they would be challenging instances for the neural network to classify.

Note that the third data set is rather different from the previous two. The third set, albeit having a larger sample size, is non-exhaustive, and graphs in the set may be isomorphic to each other. The graphs in the third set are much more likely to be 'hard' instances for the classification problem, whereas the other two data sets have more easy instances, say, very sparse graphs or very dense graphs. Since the third data set is more difficult, it is expected that those NN models might not attain the same level of performance as the models for the previous two data sets.

For our experiments, we always pass the binary variable

$$x(G) = \begin{cases} 1: & \text{there is a length-k cycle in } G \\ 0: & \text{otherwise} \end{cases}$$

as labels. We use different choices of information of graph as input features, and for each combination of data set and input feature selection, we build a FNN and RNN model for it.

All the NNs have the following architecture specifications and hyperparameters, unless otherwise specified:

- Hidden layers all have a dropout rate of 0.2;
- Hidden layers are all L^2-regularized with penalty rate 0.0003;
- Output layer has two nodes with *softmax* as activation function, with sparse categorical cross-entropy as loss function; and
- The optimizer algorithm is Adaptive Moment Estimation (ADAM), with learning rate 0.001 and decay rate 10^{-5}.

Other details and specification of the NNs, such as the types, numbers and sizes of hidden layers, differs for every experiment, and are addressed separately in each experiment section. We also report the test set accuracy of each trained NN model. Listed below are experiments, categorized by the type of input features.

3.2. Adjacency matrix as input feature

The adjacency matrices are passed as input features. About one-tenth of the data is reserved as testing data, and the rest is training data. The table below is the hyperparameters and performance of the NNs.

NN Type	Data Set	Inner Layer Width	# LSTM layers	# Dense Layers	Accuracy
FNN	8-graphs	16	0	4	93%
RNN	8-graphs	16	2	1	96%
FNN	9-graphs	16	0	4	93%
RNN	9-graphs	16	2	2	95%
FNN	20-graphs	32	0	4	76%
RNN	20-graphs	32	2	2	82%

TABLE 1. Architecture and performance of neural network models with adjacency matrix of graphs as input features.

For 'small' graphs such as 8- and 9-graphs, the NNs successfully learnt to determine whether they have 8-cycle(s) or not. The performance is comparatively lower for the models with 20-graphs as data set. The performance drop is expected as the 20-graph data set has more 'hard' instances for the classification problem. Also, RNN performs better than FNN for every data set by 2 to 4%.

These neural networks are rather compact, as each of them have four hidden layers with only 16 to 32 neurons. We also tried training NNs with wider layers of up to 256 neurons per layer, but that only led to quicker overfitting while providing no further increase in accuracy. Given that a simpler and larger network can both achieve the same level of accuracy, we favor the simpler one.

3.3. Eigenvalues of adjacency matrix as input features

The eigenvalues of the adjacency matrices of the graphs are passed as features. Unlike the adjacency matrices, eigenvalues do not uniquely identify a graph, resulting in a loss of information. Although one can theoretically extract eigen-information from the adjacency matrices alone, and the main objective of NN is to let the machine figure out the best latent embedding for a given classification problem, we nevertheless fed the eigenvalues to the NNs.

NN Type	Data Set	Inner Layer Width	# LSTM layers	# Dense Layers	Accuracy
FNN	9-graphs	16	0	4	88%
RNN	9-graphs	16	2	2	90%
FNN	20-graphs	32	0	4	76%
RNN	20-graphs	32	2	2	79%

TABLE 2. Architecture and performance of neural network models with eigenvalues of graphs as input features.

Without regularization, the models predict most of the input data as 'true' instances. It is possibly due the model's over-reliance on the first eigenvalue, which give a rough estimate of the average- and maximum-degree of a graph. With L^2 regularization, a ridge penalty is applied to the parameters of nodes in the inner layers, thus reducing the reliance on a single feature. After regularization, the models all perform much better.

Using only the eigenvalues as input features, the model performs decently and compromises only about 5% accuracy, compared to using adjacency matrices instead. In fact, for the 20-graph FNNs, the eigenvalue model performs just as well as the adjacency matrix counterpart. Therefore eigenvalue proves to be a very useful feature for such classification problems, especially when the eigenvalue-vector is much more compact ($\mathbb{R}^{n\times 1}$) than the adjacency matrix ($\mathbb{R}^{n\times n}$).

3.4. Adjacency matrix and its eigenvalues as input features

Given the previous experiment's relative success, we would also want to see if eigenvalues extract and provide any additional information to the NNs. Therefore, both eigenvalues and adjacency matrices are used as input features in this experiment. The eigenvalues are appended as the $(n+1)^{\text{th}}$ row to the n-by-n adjacency matrices, then these n-by-$(n+1)$ matrices are passed as input features.

NN Type	Data Set	Inner Layer Width	# LSTM layers	# Dense Layers	Accuracy
FNN	9-graphs	16	0	4	91%
RNN	9-graphs	16	2	2	93%
FNN	20-graphs	32	0	4	74%
RNN	20-graphs	32	2	2	81%

TABLE 3. Architecture and performance of neural network models with adjacency matrices and eigenvalues of graphs as input features.

These models perform a bit worse than their counterparts with only adjacency matrices as features, but better than only using eigenvalues as features. Like the previous experiment, NNs are significantly over-reliant on graph density, causing most graphs to be classified as true-instances. However, L^2 regularization does not seem to work on these models.

3.5. Using different datasets for training and testing

Here, we pass adjacency matrices as input features, but the testing and training data come from different data sets.

NN Type	Training Set	Testing Set	Layer Width	# LSTM	# Dense	Accuracy
FNN	8-graphs	9-graphs	16	0	4	88%
RNN	8-graphs	20-graphs	64	2	2	54%

TABLE 4. Architecture and performance of neural network models with adjacency matrices of graphs as input features. Note that these models' training and testing data come from different data sets.

Training on 8-graphs and testing on 9-graphs, the FNN is good at identifying true-instances (95%), but does not work as well on identifying false instances (77%). After L^2 regularization, the NN stopped over-classifying dense graphs as true instances.

	Predicted False	Predicted True	%
Actual False	2521	125	95%
Actual True	767	4053	84%
	77%	97%	88%

TABLE 5. Confusion matrix of the FNN model prediction reults.

3.6. Summary

Neural networks have good overall performance on all data sets, attaining up to 95% accuracy on small graphs, and 80% accuracy on larger graphs of harder instances. There may be a graph representation or some selection of features that are more suitable for the NNs to learn on the harder instances.

The set of eigenvalues of graph is a very telling factor on the graph's topology. Adjacency matrix provides full information of the graph, while eigenvalues do not uniquely identify a graph. Nevertheless, using eigenvalues alone as features, our models compromised less than 5% accuracy. For large graphs, NNs achieve the same level of accuracy with only eigenvalues as features as compared with adjacency matrices.

Density of graph is (unsurprisingly) a significant factor in this classification problem. When unregulated, NNs tend to over-rely on density as a predictor for the classification. Before applying L^2 regularization to the loss function of the neural networks, the NNs tend to over-classify dense graphs as positive-instances. Although L^2 regularization did not improve the accuracy by much, it alleviated the problem of over-reliance on graph density, as seen in the confusion matrices of the testing result.

Surprisingly, recurrent NNs with LSTM layers improve the NN's performance, as compared to FNNs. While RNN is conventionally used for temporal or continuous data, such as video, music and speech, the LSTM architecture may have more computational power, and therefore performs better on classification problems not necessarily temporal - such as this problem.

Overall, the models perform well on small graphs, but the models on large graphs and attempts to enforce generalizations could be further improved. The models on harder instances large graphs are not immediate successes, perhaps due to the discrete and combinatorial nature of the neural network.

Bibliography

1. Norman Biggs, *Algebraic graph theory*, Cambridge University Press, 1996.
2. Facundo Bre, Juan Gimenez, and Víctor Fachinotti, *Prediction of wind pressure coefficients on building surfaces using artificial neural networks*, Energy and Buildings **158** (2017).
3. Nathaniel Dean, Linda Lesniak, and Akira Saito, *Cycles of length 0 modulo 4 in graphs*, Discrete Mathematics **121** (1993), no. 1, 37 – 49.
4. William Giovinazzo, *Overfitting / underfitting – how well does your model fit?*, https://meditationsonbianddatascience.com/2017/05/11/overfitting-underfitting-how-well-does-your-model-fit/, 2017.
5. Christopher Karl Heckman and Roi Krakovski, *Erdös-gyárfás conjecture for cubic planar graphs*, Electronic Journal of Combinatorics **20** (2013).
6. Sai Nikhilesh Kasturi, *Underfitting and overfitting in machine learning and how to deal with it*, https://towardsdatascience.com/underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6fe4a8a49dbf, 2019.
7. Klas Markström, *Extremal graphs for some problems on cycles in graphs*, Congressus Numerantium **171** (2004).
8. David W. Matula, *The employee party problem*, Notices of the American Mathematical Society **19** (1972), 382.
9. thomassen C., (1983).
10. Pedro Torres, *Deep learning: Recurrent neural networks*, https://medium.com/deeplearningbrasilia/deep-learning-recurrent-neural-networks-f9482a24d010, 2018.
11. Zhe Wang, Tong Zhang, and Yuhao Zhang, *Distinguish hard instances of an np-hard problem using machine learning*, 2013.
12. David Warde-Farley, Ian J. Goodfellow, Aaron Courville, and Yoshua Bengio, *An empirical analysis of dropout in piecewise linear networks*, 2013.
13. Virginia Vassilevska Williams, *Cs 267 lecture notes 3*, Stanford University, 2016.

APPENDIX A

Python code

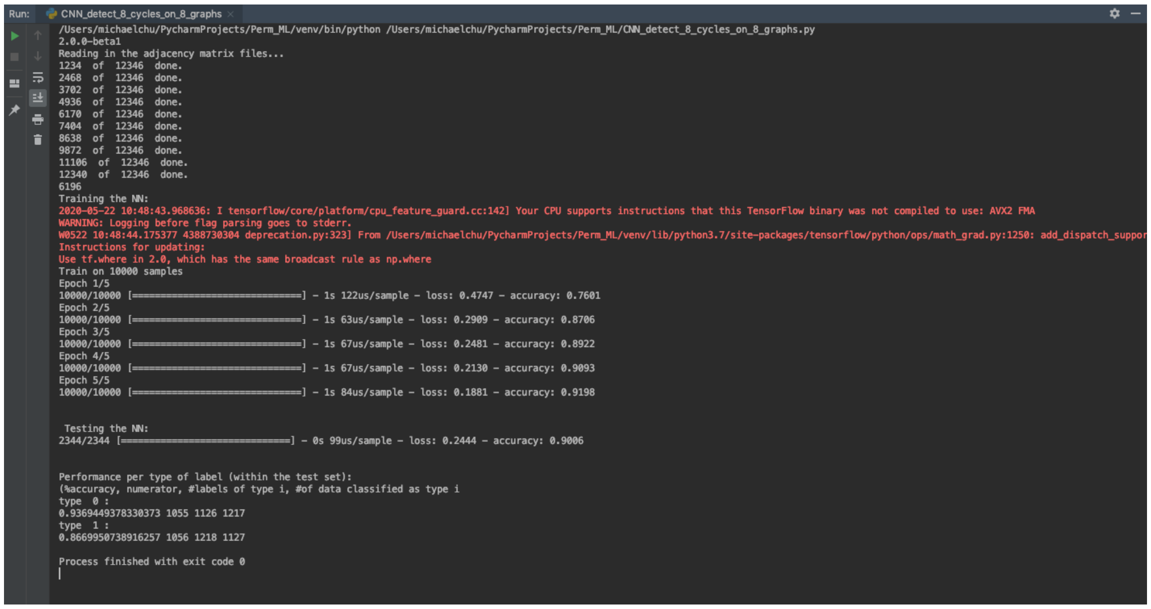
In this section, I present a selection of Python codes used in the experiments. Note that the experiments use Tensorflow, a comprehensive package for deep learning.

```
import numpy as np
import time
import itertools
import csv
from sklearn.metrics import confusion_matrix

import networkx as nx

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, LSTM
print(tf.__version__)

adj_mat_rel_path = 'v8_adj_mat/'
adj_mat_prefix = '8_'
adj_mat_suffix = '_adj_mat.txt'

N_start = 3
N_end = 12346
N = N_end    N_start + 1
num_V = 8

training_size = 10000
test_size = (N)    training_size

def main():
    # loading in the data
    data = np.zeros((N, num_V, num_V))
```

```
print('Reading in the adjacency matrix files...')
for i in range(N_start, N_end + 1):
    temp = np.loadtxt(adj_mat_rel_path + adj_mat_prefix + str(i) + adj_mat_suffix)
    data[i  3, :, :] = temp
    if i % int(N_end / 10) == 0:
        print(i, ' of ', N_end, ' done.')

cycle_length_data_8 = np.loadtxt('cycleLengthData_8.txt', skiprows=2)
# labels each graph as 1 if it has an 8 cycle, and 0 otherwise
labels = (cycle_length_data_8[:, 5] >= 1) * 1

# shuffling the data and labels in unison, while preserving the shuffling indices
shuffle_index = np.array([i for i in range(N)])
np.random.shuffle(shuffle_index)
training_index = shuffle_index[0:training_size]
test_index = shuffle_index[training_size:N]

training_data = data[training_index, :, :]
test_data = data[test_index, :, :]
training_labels = labels[training_index]
test_labels = labels[test_index]

print(sum(labels))

# compiling, then evaluating, the NN

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    # Adds dense layers with 64 units to the model:
    layers.Dense(64, activation='relu'),
    layers.Dense(64, activation='relu'),
    layers.Dense(64, activation='relu'),
    layers.Dense(64, activation='relu'),
    # Add a softmax layer with 2 output units:
    layers.Dense(2, activation='softmax')])


model.compile(optimizer=tf.optimizers.Adam(lr=0.001, decay=1e 5),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

print("Training the NN:")
model.fit(training_data, training_labels, epochs=5)

print("\n\n Testing the NN:")
result = model.evaluate(test_data, test_labels)

class_bin = model.predict(test_data)
classification = np.argmax(class_bin, axis=1)
```

```
    print("\n\nPerformance per type of label (within the test set):")
    print("(%accuracy, numerator, #labels of type i, #of data classified as type i")
    for i in range(2):
        print("type ", i, ":")
        denom = sum(test_labels == i)
        lbl_eq_i = [test_labels[j] == i for j in range(len(test_labels))]
        lbl_eq_class = [test_labels[j] == classification[j] for j in range(len(test_labels))]
        numer = sum([lbl_eq_i[j] and lbl_eq_class[j] for j in range(len(test_labels))])
        print(numer / denom, numer, denom, sum(classification == i))


if __name__ == "__main__":
    main()
```

FIGURE 1. Python code for compiling and running a feed-forward neural network on the 8-graphs dataset



```
/Users/michaelchu/PycharmProjects/Perm_ML/venv/bin/python /Users/michaelchu/PycharmProjects/Perm_ML/CNN_detect_8_cycles_on_8_graphs.py
2.0.0-beta1
Reading in the adjacency matrix files...
1234  of  12346  done.
2468  of  12346  done.
3702  of  12346  done.
4936  of  12346  done.
6170  of  12346  done.
7404  of  12346  done.
8638  of  12346  done.
9872  of  12346  done.
11106  of  12346  done.
12340  of  12346  done.
6196
Training the NN:
2020-05-22 10:48:43.968636: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
WARNING: Logging before flag parsing goes to stderr.
W0522 10:48:44.175377 4388730304 deprecation.py:323] From /Users/michaelchu/PycharmProjects/Perm_ML/venv/lib/python3.7/site-packages/tensorflow/python/ops/math_grad.py:1250: add_dispatch_suppor
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
Train on 10000 samples
Epoch 1/5
10000/10000 [==============================] - 1s 122us/sample - loss: 0.4747 - accuracy: 0.7601
Epoch 2/5
10000/10000 [==============================] - 1s 63us/sample - loss: 0.2909 - accuracy: 0.8706
Epoch 3/5
10000/10000 [==============================] - 1s 67us/sample - loss: 0.2481 - accuracy: 0.8922
Epoch 4/5
10000/10000 [==============================] - 1s 67us/sample - loss: 0.2130 - accuracy: 0.9093
Epoch 5/5
10000/10000 [==============================] - 1s 84us/sample - loss: 0.1881 - accuracy: 0.9198


 Testing the NN:
2344/2344 [==============================] - 0s 99us/sample - loss: 0.2444 - accuracy: 0.9006


Performance per type of label (within the test set):
(%accuracy, numerator, #labels of type i, #of data classified as type i
type  0 :
0.9369449378330373 1055 1126 1217
type  1 :
0.8669950738916257 1056 1218 1127

Process finished with exit code 0
```

FIGURE 2. A screenshot of the model training process.
First, the data loading its counted down (it may take a bit of time to load in all the data for large data sets). Then the model's loss and accuracy is reported per epoch. After the NN finishes training process, its performance is tested on the test set.
The model's performance per label type is also reported. For example, in the screenshot, the model achieves 93% true-negative rate. Out of 1126 data that are labelled as real-negatives, the model is able to identify 1055 of them. The model predicts a total of 1217 negatives, which means there are $1126 - 1055 = 162$ false-negatives in our prediction.