COSC470 Research Project Report

An AI System for Automatic Grapevine Pruning Decision Making

Marvin Goesmann
Supervised by Dr. Richard Green

2020

Abstract

Vine pruning is a commonly manual process in vineyards requiring costs and time and requiring skill and expertise from pruners. Over the years, advancements made in the related fields of AI has provided the potential to automate this process and aim to increase grape yield while reducing costs. This paper focuses on one specific step of the process, in which an AI classifies vines to be either pruned or laid down. We introduce a neural network architecture involving a graph structure to encode the vine sequence to overcome limitations in past research relating to lack of encoding structural information. By training with a simple and fairly short dataset, the system is able to learn simple pruning rules through training examples and managed to achieve a 97.4% classification accuracy when evaluated by a dataset from the same source as the training data, and 93.5% when evaluated by a more complex and unknown dataset, proving it is able to learn unknown features and can generalise to structures more complex than what is was trained on.

Acknowledgements

Dr Richard Green – Supervisor
Oliver Batchelor – Project assistance and mentoring
Jaco Fourie – Source of expertise and main network designer
Maaratech – Project resources

Contents

1. Introduction

1.1 Background

Agriculture is an expensive and essential part of our economy that takes up a large amount of manual labour and time. The shift is slowly trending towards automating aspects of the agriculture process as technology becomes more advanced and cheaper to utilize. The introduction and improvement of computer vision systems have started to allow us to tackle the complexity of crop production and all its variable parameters, and along with advancements in Artificial Intelligence and Neural Networking, can introduce highly complex analysis and decision making on the observed data.

Of these processes in crop production, there is monitoring, watering, harvesting, and pruning, along with many more depending on the kind of crop. Pruning is the removal of dead, overgrown or unnecessary branches, stems or twigs to promote growth, prevent disease and maximize yield. In vineyards, this occurs annually during winter in its dormant season, and can make up large costs and time commitments. There are many different methods to manual vine pruning, some which are more complex than others. The report compares the various methods, and by addressing the issues, we attempt to translate vine rules into an AI system to make these decisions automatically.

1.1.1 Structure

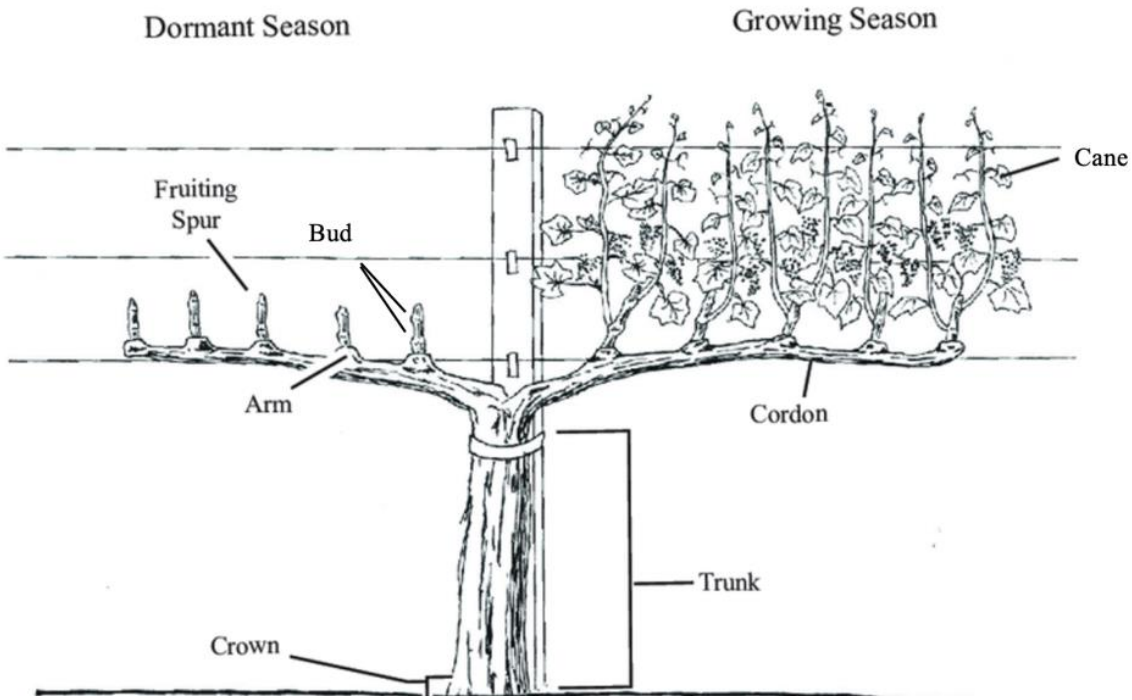

Figure 1 – Structure of a grapevine. https://www.researchgate.net/figure/Grapevine-structures-and-features-self-rooted-vine-Drawing-by-Scott-Snyder_fig3_237296145

A grapevine consists of many parts. In order to describe the pruning process, some of these terms need to be addressed. A trunk holds the grapevine up with two to four horizontal cordons branching out from the top end of the trunk. Each of cordons contains a set of canes before being pruned, or a set of spurs containing buds in which new canes will grow from. Canes become cordons after being tied down, from which more canes will grow from.

1.1.2 Pruning Strategies

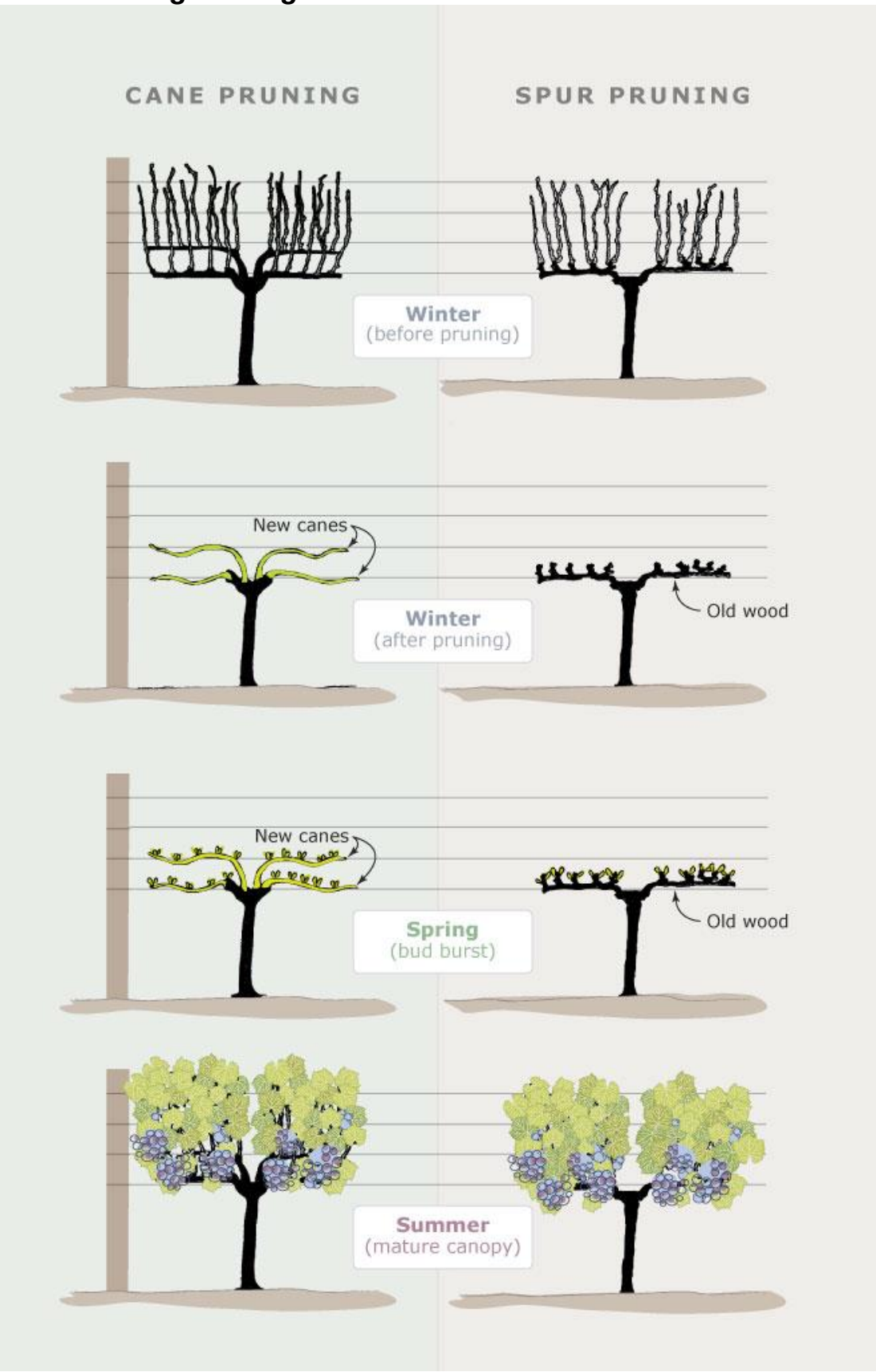

Figure 2 - Cane Pruning vs Spur Pruning https://teara.govt.nz/en/diagram/18314/cane-and-spur-pruning

Two common pruning strategies are cane pruning and spur pruning. The differences in these strategies can affect yield, plant life and disease resistance.

Spur Pruning
Spur pruning is the more simple method for pruning grapevines. There is no selection criteria as all canes are pruned from the cordons, generally leaving up to 6 buds on the grapevine. By locating the main cordons, everything above the recommended amount of buds is removed through pruning. Typically, the amount of buds left is about 2-4. Spur pruning is typically used to reduce time and costs, or with vines which produce fruit on new growth coming from buds close to the base of canes, close to the trunk.

Cane Pruning
Cane pruning is a more complex and expensive process. This involves selecting certain canes that are chosen to be most optimal. Typically 4-6 canes are chosen, where two are cut to about two buds while the rest are lain to form new bearing canes which will eventually become cordons. This process requires an intuitive sense from the pruner, who has to make a decision based on their own experience and training. This is the pruning method we try to emulate through our AI system.

1.2 Prior Work

Prior work has been attempted to try to automate this process, and their shortcomings will be evaluated and attempted to be improved on throughout this research.

1.2.1 Classification and Search based Pruning

Botterill et al. [1] demonstrates a most recent case of a full system that is able to successfully prune grapevines. The paper describes the full process from robot traversing, imaging, modelling, deciding and finally cutting. The decision process is outlined as an AI system that decides on canes to keep through a pruning scheme. The pruning scheme contains a feature vector of cane attributes (length, position, angle, distance below wires and where the cane grows from) which is fed to a cost function which is described as a simple linear combination of features. Every vine is compared with each pruning scheme where the highest scoring is selected.

The cost function of the AI system above is based on research by Corbett-Davies et al. [2]. The study researches both classification algorithms and search algorithms to make decisions for pruning grapevines. The main classification algorithm utilised a decision tree to classify canes by pruning rules determined by viticulture experts, but was deemed unsuccessful with a less than 50% pruning success rate. This was identified as being due to the limitation of not being able to classify canes in isolation, which is essential when deciding on which canes to prune. As its alternative, the main search algorithm that was developed was the same brute-force search mentioned in the study by Botterill et al. The cost function, again, is a simple weighted sum of features mentioned above along with a number missing feature which is global to the pruning scheme. The weightings were determined by training through a simple global optimisation problem.

The biggest reason for the lack of success for the classification approach was because canes cannot be classified in isolation. Therefore a system needs to be created that observes the entire vine or group of canes, which is what the search-based approach attempted to solve. A limitation is approaching an expert to determine important information and pruning parameters about the plants, which was researched by Saxton et al. [3] when attempting to translate expert rules into AI decision making. Firstly, this can be expensive and time-consuming if the expert opts to sort through a lot of examples to fully demonstrate their decision making. In an alternate case of just describing decision making without examples, this introduces unreliability through communication issues as areas like this are very difficult to describe. Moreover, both systems do not take into the order into account in which the canes are pruned. Similar to the relationships between canes, the decision of which order to prune affects the other canes on the vine, so this should be acknowledged.

1.2.2 Spur Pruning

Many studies have attempted spur pruning, but have fallen into the same limitations that spur pruning offers in general. Gao and Lu [4] offer an approach to spur pruning where all canes are cut down to a small shoot with two buds, where there is no cane decision making. It had an 85% success rate from 10 examples, but mentioned limitations of their system working on 2D images instead of 3D models which introduces issues when certain views are obstructed. On top of the mentioned limitations, we have determined that spur pruning is less advantageous to cane pruning, so this approach was dropped.

1.2.3 Fuzzy Multicriteria Decision-Making

Tisseyre et al. demonstrates a fuzzy multicriteria decision-making approach to grapevine [5]. This method effective at reasoning about uncertainty and inaccuracy, which is often the case when basing it off expert preference. The paper is fairly outdated being published in 1997 and many of its applications have been replaced with the advancements in machine learning, although it still provides some insights into the challenges and considerations that need to be applied when designing such a system.

1.2.4 Other Plants

Pruning decision making has been attempted in other plants, including various trees ([6], [7], [8], [9], [10], [11], [12], [13]) and tomato plants [14] involving manual and automated strategies. In general, the studies involve very simple pruning schemes because of the nature of their plant, for example taking measurements and comparing to a fixed base value or ratio to determine pruning decision [9], [13], [14]. A simple system do not apply well to grapevines because of the greater complexity of pruning decision making.

1.3 Aims and Objectives

By combining limitations from past research and recommendations from experts, some objectives are addressed to guide the design and implementation of the system. These are:

1. The system is simple to train.
2. The system needs to consider pruning order.
3. Classifying canes needs to consider its relation to other canes.
4. The system should pick up unseen characteristics in the training data.

The system needs to be simple to train in that it does not require an expert pruner to label thousands of training data, or that large amounts of real life data needs to be acquired to train. The system needs to consider pruning order as the classification and pruning of one cane affects the decision of the next cane. The classification of canes needs to consider its relation to other canes because the positions of canes on the cordon, distance between canes and other relationships needs to be considered in the scheme. Finally, the system needs to be more intelligent than simple static pruning rules or only support the specific training data it had been fed, instead it should be able to account for variation and generalise for structures that are unseen or more complex than the training set.

2. Design and Implementation

2.1 Graph Neural Networks

In order to achieve our aims, a neural network approach was adopted. Deep learning through neural networks allows us to teach our system how to reason about decision making through training. However, standard network architectures as a set of features will not allow us enough functionality to model both canes and the vine as a whole. Because of this, we use graph neural networks to encode relationships between our canes.

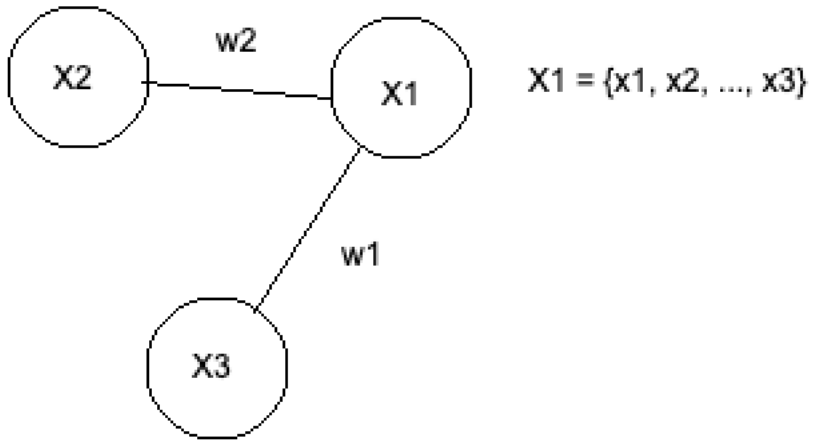

Figure 3 - Graph with nodes, edges and node features

Graph approaches is an option to represent our vines. Such a graph contains a set of nodes connected by weights, where each node consists of a set of features. Neural networks can be adapted in a way to benefit from the way graphs encode important data and relationships.

Graph neural networks is a type of neural network which operates on a graph structure. Through graph neural networks, we can represent nodes as cane features and weights as the relationship between canes.

2.2 Vine Graph Representation

Vines are represented as a graph where nodes are feature vectors of cane parameters and edges are relationships between the canes. Consultation with vine pruning experts allows us to realise the important parameters of a vine that factors into the decision making process.

The nodes are feature vectors which are comprised of:

- Cane diameter
- Cane length
- Number of buds on the cane
- Horizontal distance from cane base to vine trunk centre
- Vertical distance from cane base to bottom trellis wire

The edges between nodes is based on the Euclidean distance between cane bases, where canes on opposite sides of the trunk have no weight. Edges with a smaller distance have a greater weight to represent a greater impact between the canes.

Cane diameter and length are the simplest parameters to describe a cane and base pruning decision making on. The number of buds on the cane is also important as they determine where new canes grow when it is laid down as a bearer. Finally, splitting distance into horizontal and vertical counterparts was determined to be more effective than having just one distance, as both components have different consequences and therefore affect decision making in different ways. These are measured from the base of the cane because it is more useful in the decision making process than measuring from the tip or centre.

We also determined that cane distances between each other was the primary relationship that would affect decision making. An equal amount of canes are typically kept on each side of the trunk, so these weights are ignored, therefore resulting in canes with relationships to all of the other canes on its side.

2.3 Graph Attention Networks

Graph Attention Networks (GATs) are novel neural network architectures that operate on graph structures through masked self-attentional layers [15] to overcome limitations of prior graph neural networks ([16], [17], [18]) [19].

Typical CNNs are very successful with types of data like images where it can utilise the regular connectivity between data (such as pixels connected to neighbouring pixels), making it simple to design things like small kernel matrices to slide over and operate on the data. In graph structures, it is more difficult to generalise a convolutional operator because of the irregular structure of the data. Ideally, we would like our graph convolutional layers to have the same ideal properties of something more structured like image convolutions, being computation and storage efficiency, fixed number of parameters, localisation, ability to specify arbitrary

importance to different neighbours, and applicability to inductive problems (arbitrary, unseen graph structures) [19].

A graph convolutional layer computes a set of new node features based on the input features as well as the graph structure [19]. The graph convolutional operator is an aggregation of features across neighbourhoods through self-attention over the node features. A graph attention layer takes the input features and adjacency matrix and calculates the attention coefficients after a linear feature transformation. The graph structure is injected by masking the coefficients by the weights matrix, which are normalised with softmax activation. Regularisation can be applied using dropout, and a linear feature transformation applies the input features with the new attention coefficients.

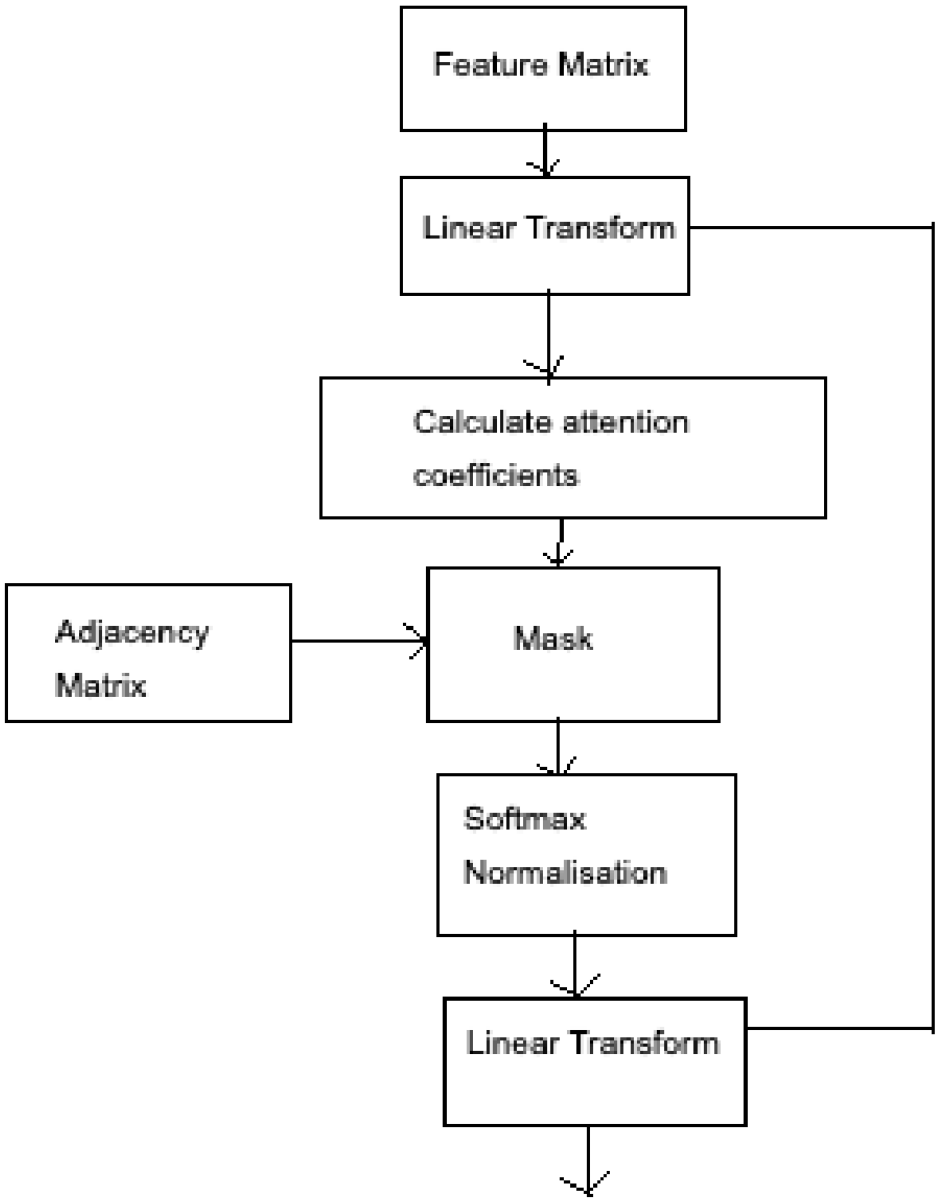

Figure 4 - A GAT Layer

In our network, the attention coefficients represent the impact of neighbouring cane features on the classification of another cane. This provides localisation by comparing canes in the local neighbourhood, thus ignoring canes which will not have an impact on the cane being classified. It can also generalise to unseen nodes in the graph meaning that it should be able to generalise to any vine structure, not just the training data. Computation and storage efficiency is provided through parallelisation and the use of simple matrix operations [19].

2.4 Recurrent Neural Networks

While Graph Attention Networks helps us solve our aims 3 and 4, we now look at achieving our second aim: *The system needs to consider pruning order*. This can be achieved by encoding an order into our network between canes.

Recurrent Neural Networks are a class of neural networks where connections between nodes form a directed graph along a temporal sequence. RNNs process examples one at a time, retaining a memory that reflects an arbitrarily long sequence of data, allowing them to be effective in domains such as time series prediction, video analysis and music information retrieval [20].

The long short-term memory architecture (LSTM) is an RNN architecture designed to deal with difficulties in training an RRN, mainly being the vanishing gradient problem that can be encountered when training traditional RNNs [21], [22]. They are composed of a cell which remembers values over arbitrary time intervals and an input gate, output gate and a forget gate, which all regulate the flow of information in and out of the cell.

2.5 Our Network

In general, the graph structure describes the vine, while GAT is used to learn features of the vine for pruning decision making, and LSTM RNNs takes the sequence into account.

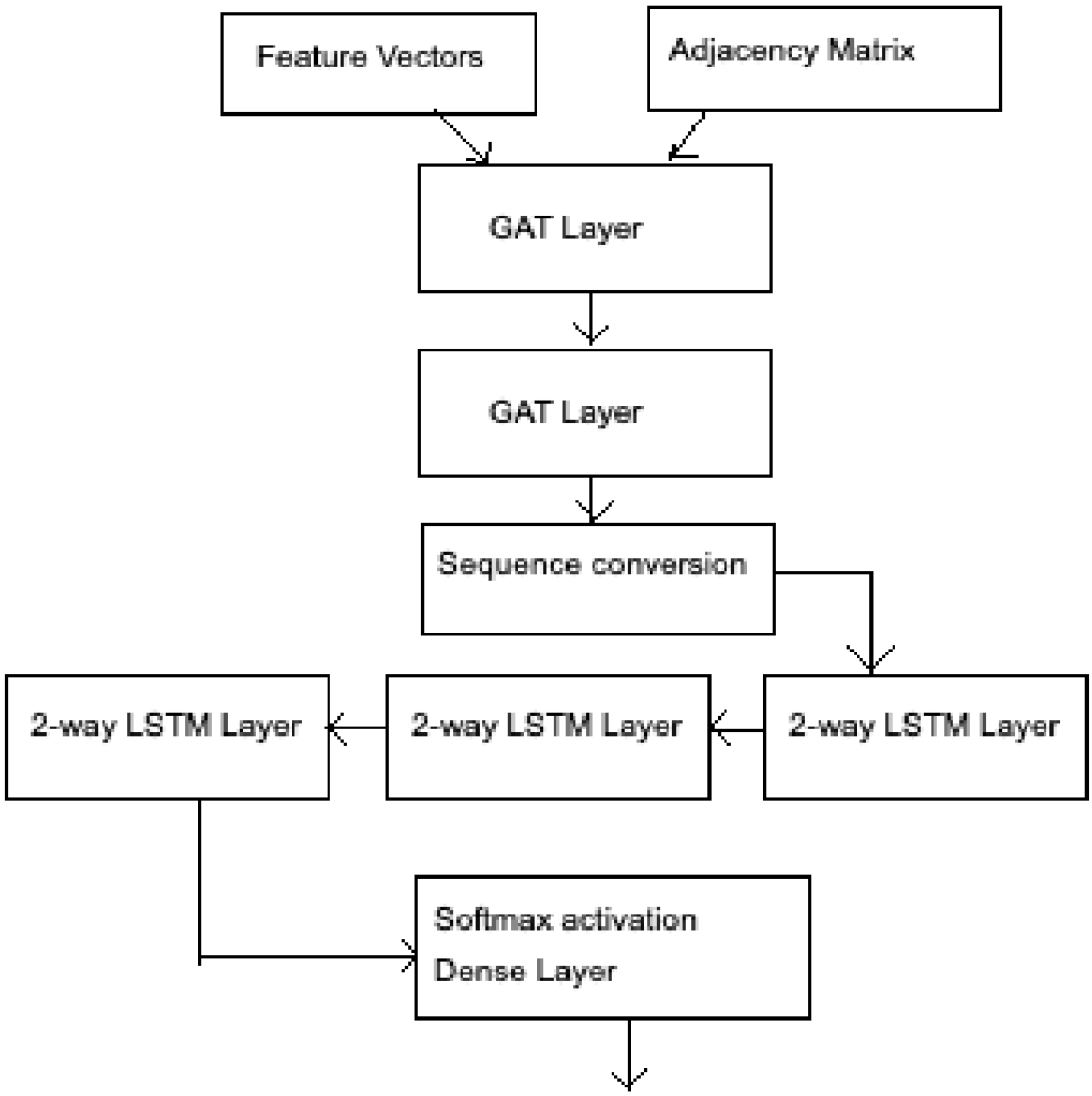

Figure 5 - Network Architecture. The sequence conversion turns output from the GAT layer into sequences of data which make their way through the LSTM layers and into the final layer.

Two GAT layers are chosen to increase the networks capacity to learn generic vine structures, where the first takes the feature vectors of the vine (nodes) and the adjacency matrix (edges) and passes its output to the next one.

Bidirectional LSTM cells are created by joining two LTSM cells where one processes the sequence forwardly and the other in reverse, to improve performance of our network.

Softmax activation is added to a final dense layer which determines the classifier output. Dropout layers are used to decrease over-fitting.

2.6 Training

In order for the network to learn sufficient features to make accurate decisions, several thousands of labelled training data often need to be provided.

Using real training data would require taking meticulous measurements from thousands of canes, observing an expert pruner to make pruning decisions and recording the result. This process is time consuming and expensive but is recommended in order to make the most accurate decisions in a real world scenario.

2.6.1 Generated Vines

An option to reduce the amount of real training data is to pre train a network with synthetic data. This is inexpensive and quick to generate, and has an infinite limit. Furthermore, there are no time schedule or seasonal constraints, as would be required with real data. This can be later fine-tuned with real data, as synthetic data can be sufficient in learning the majority of decision making, as long as the synthetic data is close enough to real. The measurements of 100 real vines were taken, and through a statistical analysis of the different features, we are able to generate realistic distributions of vine measurements.

To generate the labels, simple pruning rules are determined. The sequence of rules follow:

1. Choose exactly two bearer canes to lay and two renewal spurs to keep, where there is at most one bearer cane on each side of the vine.
2. Choose renewal spurs first and then bearer canes.
3. Canes with lower vertical distance have priority over those that are higher up.
4. Chosen canes should not be closer than 30mm from each other.

2.6.2 Realistic Synthetic Vines

Our crude generated vines are sufficient for pre-training, but we can improve our system with better data. Again, we can opt to take real data from vineyards and expert pruners, however, we opt for another synthetic method to gain the benefits that come when generating data.

SpeedTree is a group of vegetation programming and modelling software products that generates realistic virtual vegetation for animations, architecture and video games. A script is created to turn the models into a graph, in the same format as our own generated vines.

3. Results

3.1 Generated Vines

1000 vines were randomly generated and labelled based on our rules above. Of the 1000, 900 were used for a training set and the remainder as a validation set. Training and validation was computed with a NVIDIA GTX 1080 Ti graphics card.

After a 49 hour training period, our system managed to classify 97.4% of the canes in the validation set correctly.

This metric is based on percentage of canes successfully classified, and since vines may have up to 30 canes and we only choose 4 to keep, the percentage of correctly classified vines are generally high despite the canes chosen.

When using a stricter metric that counts correct classification as when all canes of the vine are classified correctly, the system gained a 66% perfect classification accuracy.

This number may look low, and does not give all of the context either. We define a final metric as a "one-error" classification accuracy, where success is measured when there is at most two classification errors (as these two classifications are swapped and therefore represent one error). The one-error classification accuracy was 96% if we add the perfect classifications to the number.

3.2 Realistic Vines

Using SpeedTree, a further 100 vines were generated, converted into graphs and labelled with our simple pruning rules. The system scored a classification accuracy of 93.5%, with a 31% perfect classification accuracy. The one-error classification accuracy was 89%.

3.2 Summary of Results

	Generated Data	**SpeedTree Data**
Classification Accuracy	97.4%	93.5%
Perfect Classification Accuracy	66%	31%
"One-error" Classification Accuracy	96%	89%

Figure 6 - Classification Accuracy Results

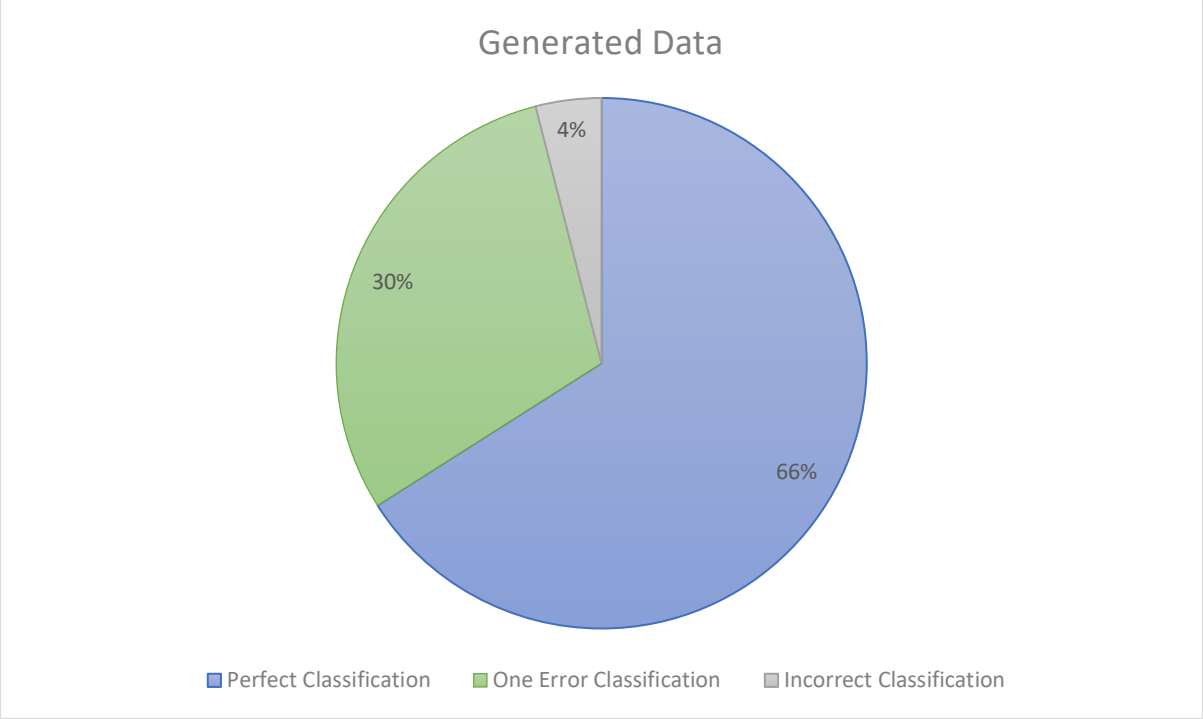

Figure 7 - Generated Data Classification

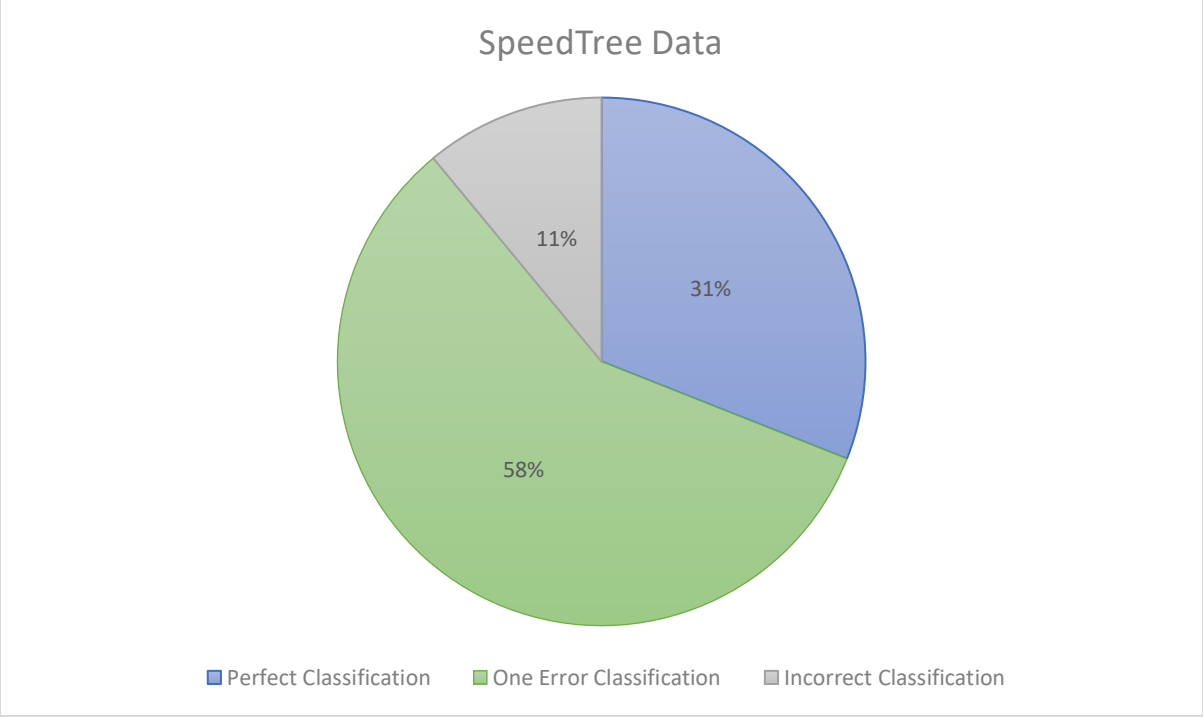

Figure 8 - SpeedTree Data Classification

4. Discussion

Using our network design and training with generated data labelled by simple pruning rules, the model was able to achieve a 97.4% classification accuracy on a dataset that was generated by the same system that generated the training set, and a 93.5% classification accuracy on a dataset generated by a completely different system. 66% of our generated vines had perfect classification, while only 31% of the realistic synthetic vines managed perfect classification. With our more fair “one-error” metric, 96% of our generated vines were pruned correctly, compared to 89% of the realistic vines.

The results show that the system was able to successfully learn the simple pruning rules entirely by training. When using data from the same source as the training data, it was able to achieve a high success rate in all three metrics. This is expected as the simulated data is fairly rigid as it is based on distribution from 100 real example measurements, thus not resulting in any unknown or unexpected data. While training took a fairly long time, this is only required once and so it does not impact the feasibility of the system.

As our system was trained on a different generator compared to SpeedTree, it makes sense that the classification accuracy decreases between the two. Furthermore, SpeedTree introduces more variety into the kinds of structures that can be generated, which introduces more chances of misclassification. Despite this, the system was able to perform well even with the SpeedTree dataset, showing the system generalises well to unknown or more complicated structures.

This also shows that pre-training with very simple rules and a fairly small dataset are sufficient to allow the network to learn rules without having to encounter a majority of possible vine structures. On top of this, it shows that simple training data can still allow the system to generalise to more complicated structures. The majority of learning can be achieved by keeping data simple, and then the remaining learning should be achieved through fine tuning with a smaller dataset of real vines pruned by an expert. This will balance time and cost of training with the effectivity of the AI at replicating pruning decisions.

The system achieves our four aims. It is simple to train because of the availability of generating data, it considers pruning order through the characteristics of RNNs with LSTM architecture, it considers its relation to other canes as it is encoded as a graph and it can pick up unseen characteristics in the training data through GAT’s ability to be applied to inductive learning problems. Therefore, it overcomes limitations in prior research such as the lack of classifying canes in isolation or in order, and the cost to train the system, both which are limitations of work by Botterill et al. [1] and Corbett-Davies et al. [2].

4.1 Limitations

A main limitation of the system so far is that it is only trained on generated data. The generated data follows a very rigid growing scheme and surface-level pruning rules, both which are not completely accurate with how pruning works in practice. Vines in nature are likely to introduce more variety than our system that is only based on 100 observed vines. When modelling vines from real data, error is also introduced which the system currently may not optimally know how to deal with. The simple pruning rules are also not perfect in

describing how an expert reasons about vines which is why a neural network approach was used in the first place instead of teaching it simple rules. It currently serves mostly as a proof of concept and a demonstration of the majority of the work required to produce a complete AI system, however as it currently stands, it is insufficient for real use in a vineyard.

The system is also designed for a particular species of grapevine. Pruning rules change between different grapevines and trellis configurations, so models trained on one are likely to perform worse on others (as seen when applying the tested model to SpeedTree vines). Moreover, the types of features on the feature vector may need to be changed between different species (for example Euclidean distance may be more useful than splitting up into horizontal and vertical distances). Some vines even benefit more from spur pruning, so a system like this is may not be required at all.

The metric that we base our results on may introduce an idea about how the system performs, however, it does not always provide complete context. A more useful (but also subjective) metric might be having an expert pruner create a scheme to validate results (such as the 5 point system by Corbet-Davies et al. [2]). This will let us know if a misclassified cane was actually a bad choice, or just a choice that is just as good as another which was labelled.

The cases in which classification was not successful (especially when there was more than one classification error for a vine) was not formally identified, which impacts our ability to find some of the unaccounted flaws in the system.

4.2 Future Work

To continue the research, the next steps would be to apply the system with data from real vines, to fine tune and test the system. 3D information and measurements should be taken from a real vineyard and should be labelled by an expert, which will serve as a small set of training data for fine tuning and an evaluation set.

The effects of pretraining using SpeedTree data should also be investigated, as when real data for evaluation becomes available, there is no longer need for all three kinds of datasets and SpeedTree may be a better trainer than our vine generator if it manages to represent real vines better.

General aspects of the project can also be extended. The network so far is a fairly simple and so there are many different architectures to test. Adding extra parameters to the set of features of a node may increase the effectivity of decisions because more structure is encoded.

One big reason for conducting the research is to eventually combine the system with all of the components that make up a complete pruning system. This includes image acquisition, 3D modelling, robot traversal and more. This may introduce more issues and uncertainties as no system is as perfect as the simulated environment throughout this study so some adjustments may have to be made to account for it.

5. Conclusion

This paper has presented a solution to automatic pruning decision making through graph neural networks. A network that consists of a graph structure to encode the vine and canes allows the network to learn about the relationship between canes. GAT layers allows the network to efficiently learn through the graph structure and provides localisation and generalisation. LSTM cells allows the sequence of cane classification to be recognised. By generating synthetic data, we show that the majority of learning can be achieved through simple data labelled by simple pruning rules. The model classifies successfully with a 97.4% classification accuracy when evaluated by a dataset from the same source as the training data, and 93.5% when evaluated by a more complex and unknown dataset. From here, the system can be extended by applying real vine structures to fine tune and validate the model.

5. References

[1] T. Botterill *et al.*, 'A Robot System for Pruning Grape Vines', *Journal of Field Robotics*, vol. 34, no. 6, pp. 1100–1122, Sep. 2017, doi: 10.1002/rob.21680.

[2] S. Corbett-Davies, T. Botterill, R. Green, and V. Saxton, 'An Expert System for Automatically Pruning Vines', in *Proceedings of the 27th Conference on Image and Vision Computing New Zealand*, New York, NY, USA, 2012, pp. 55–60, doi: 10.1145/2425836.2425849.

[3] V. Saxton, T. Botterill, and R. Green, 'First steps in translating human cognitive processes of cane pruning into AI rules for automated robotic pruning', *http://dx.doi.org/10.1051/bioconf/20140301016*, 2014, doi: 10.1051/bioconf/20140301016.

[4] M. Gao and T. Lu, 'Image Processing and Analysis for Autonomous Grapevine Pruning', in *2006 International Conference on Mechatronics and Automation*, Jun. 2006, pp. 922–927, doi: 10.1109/ICMA.2006.257748.

[5] B. Tisseyre, N. J. B. McFarlane, C. Sinfort, R. D. Tillett, F. Sevila, and A. Carbonneau, 'Fuzzy multicriteria decision-making for long cane pruning: A system for standard and complex vine configurations', *International Journal of Intelligent Systems*, vol. 12, no. 11–12, pp.

877–889, 1997, doi: 10.1002/(SICI)1098-111X(199711/12)12:11/12<877::AID-INT6>3.0.CO;2-U.

[6] L. He and J. Schupp, 'Sensing and Automation in Pruning of Apple Trees: A Review', *Agronomy*, vol. 8, no. 10, Art. no. 10, Oct. 2018, doi: 10.3390/agronomy8100211.

[7] M. Karkee, B. Adhikari, S. Amatya, and Q. Zhang, 'Identification of pruning branches in tall spindle apple trees for automated pruning', *Computers and Electronics in Agriculture*, vol. 103, pp. 127–135, Apr. 2014, doi: 10.1016/j.compag.2014.02.013.

[8] T. L. Robinson, L. I. Dominguez, and F. Acosta, 'Pruning strategy affects fruit size, yield and biennial bearing of "Gala" and "Honeycrisp" apples', *Acta Hortic.*, no. 1130, pp. 257–264, Dec. 2016, doi: 10.17660/ActaHortic.2016.1130.38.

[9] J. R. Schupp *et al.*, 'A Method for Quantifying Whole-tree Pruning Severity in Mature Tall Spindle Apple Plantings', *HortScience*, vol. 52, no. 9, pp. 1233–1240, Sep. 2017, doi: 10.21273/HORTSCI12158-17.

[10] D. C. Ferree and W. T. Rhodus, 'Apple Tree Performance with Mechanical Hedging or Root Pruning in Intensive Orchards', *Journal of the American Society for Horticultural Science*, vol. 118, no. 6, pp. 707–713, Nov. 1993, doi: 10.21273/JASHS.118.6.707.

[11] S. J. Wertheim, 'PRUNING OF SLENDER SPINDLE TYPE TREES', *Acta Hortic.*, no. 65, pp. 173–180, Jun. 1978, doi: 10.17660/ActaHortic.1978.65.26.

[12] S. Sansavini, 'MECHANICAL PRUNING OF FRUIT TREES', *Acta Hortic.*, no. 65, pp. 183–198, Jun. 1978, doi: 10.17660/ActaHortic.1978.65.28.

[13] N. M. Elfiky, S. A. Akbar, J. Sun, J. Park, and A. Kak, 'Automation of dormant pruning in specialty crop production: An adaptive framework for automatic reconstruction and modeling of apple trees', in *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, Boston, MA, Jun. 2015, pp. 65–73, doi: 10.1109/CVPRW.2015.7301298.

[14] A. Joey, Z. Jane, and L. Bo, 'Automated Pruning of Greenhouse Indeterminate Tomato Plants', in *Proceedings of the 2nd International Conference on Vision, Image and Signal Processing*, Las Vegas, NV, USA, Aug. 2018, pp. 1–9, doi: 10.1145/3271553.3271569.

[15] A. Vaswani *et al.*, 'Attention Is All You Need', *arXiv:1706.03762 [cs]*, Dec. 2017, Accessed: Oct. 19, 2020. [Online]. Available: http://arxiv.org/abs/1706.03762.

[16] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, 'Spectral Networks and Locally Connected Networks on Graphs', *arXiv:1312.6203 [cs]*, May 2014, Accessed: Oct. 19, 2020. [Online]. Available: http://arxiv.org/abs/1312.6203.

[17] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, 'Gated Graph Sequence Neural Networks', *arXiv:1511.05493 [cs, stat]*, Sep. 2017, Accessed: Oct. 19, 2020. [Online]. Available: http://arxiv.org/abs/1511.05493.

[18] M. Defferrard, X. Bresson, and P. Vandergheynst, 'Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering', *arXiv:1606.09375 [cs, stat]*, Feb. 2017, Accessed: Oct. 19, 2020. [Online]. Available: http://arxiv.org/abs/1606.09375.

[19] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, 'Graph Attention Networks', *arXiv:1710.10903 [cs, stat]*, Feb. 2018, Accessed: Oct. 19, 2020. [Online]. Available: http://arxiv.org/abs/1710.10903.

[20] Z. Lipton, 'A Critical Review of Recurrent Neural Networks for Sequence Learning', May 2015.

[21] S. Hochreiter and J. Schmidhuber, 'Long Short-term Memory', *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997, doi: 10.1162/neco.1997.9.8.1735.

[22] A. Sherstinsky, 'Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network', *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, Mar. 2020, doi: 10.1016/j.physd.2019.132306.

6. Appendices

Graph Attention Layer

```
from __future__
import
absolute_import

import tensorflow as tf
from tensorflow import keras as k
from tensorflow.keras import activations, constraints, initializers,
regularizers
from tensorflow.keras.layers import Layer, Dropout, LeakyReLU


class GraphAttention(Layer):
    """
    This custom layer is based greatly on the layer defined by Daniele
Grattarola in his keras implementation
    https://github.com/danielegrattarola/keras-gat
    """

    def __init__(self,
                 F_,
                 attn_heads=1,
                 attn_heads_reduction='concat',        #    {'concat',
'average'}
                 dropout_rate=0.5,
                 activation='relu',
                 use_bias=True,
                 kernel_initializer='glorot_uniform',
                 bias_initializer='zeros',
                 attn_kernel_initializer='glorot_uniform',
                 kernel_regularizer=None,
                 bias_regularizer=None,
                 attn_kernel_regularizer=None,
                 activity_regularizer=None,
                 kernel_constraint=None,
                 bias_constraint=None,
                 attn_kernel_constraint=None,
```

```
                 residual_layers=False,
                 **kwargs):
        if attn_heads_reduction not in {'concat', 'average'}:
            raise ValueError('Possbile reduction methods: concat,
average')

        self.F_ = F_  # Number of output features (F' in the paper)
        self.attn_heads = attn_heads  # Number of attention heads (K in
the paper)
        self.attn_heads_reduction = attn_heads_reduction  # Eq. 5 and
6 in the paper
        self.dropout_rate = dropout_rate  # Internal dropout rate
        self.activation = activations.get(activation)  # Eq. 4 in the
paper
        self.use_bias = use_bias

        self.kernel_initializer = initializers.get(kernel_initializer)
        self.bias_initializer = initializers.get(bias_initializer)
        self.attn_kernel_initializer                                    =
initializers.get(attn_kernel_initializer)

        self.kernel_regularizer = regularizers.get(kernel_regularizer)
        self.bias_regularizer = regularizers.get(bias_regularizer)
        self.attn_kernel_regularizer                                    =
regularizers.get(attn_kernel_regularizer)
        self.activity_regularizer                                       =
regularizers.get(activity_regularizer)

        self.kernel_constraint = constraints.get(kernel_constraint)
        self.bias_constraint = constraints.get(bias_constraint)
        self.attn_kernel_constraint                                     =
constraints.get(attn_kernel_constraint)
        self.supports_masking = False
        self.residuals = residual_layers

        # Populated by build()
        self.kernels = []       # Layer kernels for attention heads
        self.biases = []        # Layer biases for attention heads
        self.attn_kernels = []  # Attention kernels for attention heads

        if attn_heads_reduction == 'concat':
            # Output will have shape (..., K * F')
            self.output_dim = self.F_ * self.attn_heads
        else:
            # Output will have shape (..., F')
            self.output_dim = self.F_
```

```
        if self.residuals:
            # add the residual layers
            self.output_dim *= 2

        super(GraphAttention, self).__init__(**kwargs)

    def build(self, input_shape):
        # Input shape from original implementation is [(None, 1433),
(None, 2708)]
        assert len(input_shape) >= 2
        F = input_shape[0][-1]

        # Initialize weights for each attention head
        for head in range(self.attn_heads):
            # Layer kernel
            kernel = self.add_weight(shape=(F, self.F_),

initializer=self.kernel_initializer,

regularizer=self.kernel_regularizer,

constraint=self.kernel_constraint,
                                     name='kernel_{}'.format(head))
            self.kernels.append(kernel)

            # # Layer bias
            if self.use_bias:
                if self.residuals:
                    bias_shape = self.F_*2
                else:
                    bias_shape = self.F_
                bias = self.add_weight(shape=(bias_shape, ),

initializer=self.bias_initializer,

regularizer=self.bias_regularizer,

constraint=self.bias_constraint,
                                       name='bias_{}'.format(head))
                self.biases.append(bias)

            # Attention kernels
            attn_kernel_self = self.add_weight(shape=(self.F_, 1),

initializer=self.attn_kernel_initializer,
```

```
regularizer=self.attn_kernel_regularizer,

constraint=self.attn_kernel_constraint,

name='attn_kernel_self_{}'.format(head),)
            attn_kernel_neighs = self.add_weight(shape=(self.F_, 1),

initializer=self.attn_kernel_initializer,

regularizer=self.attn_kernel_regularizer,

constraint=self.attn_kernel_constraint,

name='attn_kernel_neigh_{}'.format(head))
            self.attn_kernels.append([attn_kernel_self,
attn_kernel_neighs])
        self.built = True

    def call(self, inputs):
        # The assumption here is that the input contains all the nodes
from a single graph so the adjacency
        # matrix is square.
        X = inputs[0]  # Node features (N x F)
        A = inputs[1]  # Adjacency matrix (N x N)

        outputs = []
        for head in range(self.attn_heads):
            kernel = self.kernels[head]  # W in the paper (F x F')
            attention_kernel  =  self.attn_kernels[head]   #  Attention
kernel a in the paper (2F' x 1)

            # Compute inputs to attention network
            features = k.backend.dot(X, kernel)  # (N x F')

            # Compute feature combinations
            # Note: [[a_1], [a_2]]^T [[Wh_i], [Wh_2]] = [a_1]^T [Wh_i]
+ [a_2]^T [Wh_j]
            attn_for_self           =           k.backend.dot(features,
attention_kernel[0])    # (N x 1), [a_1]^T [Wh_i]
            attn_for_neighs          =          k.backend.dot(features,
attention_kernel[1])  # (N x 1), [a_2]^T [Wh_j]

            # Attention head a(Wh_i, Wh_j) = a^T [[Wh_i], [Wh_j]]
            dense             =             attn_for_self             +
k.backend.transpose(attn_for_neighs)  # (N x N) via broadcasting
```

```
            # Add nonlinearty
            dense = LeakyReLU(alpha=0.2)(dense)

            # Mask values before activation (Vaswani et al., 2017)
            dense *= A
            # mask = -10e9 * (1.0 - A)
            # dense += mask

            # Apply softmax to get attention coefficients
            dense = k.backend.softmax(dense)  # (N x N)

            # Apply dropout to features and attention coefficients
            dropout_attn = Dropout(self.dropout_rate)(dense)  # (N x N)
            dropout_feat = Dropout(self.dropout_rate)(features)  # (N
x F')

            # Linear combination with neighbors' features
            node_features = k.backend.dot(dropout_attn, dropout_feat)
# (N x F')

            if self.residuals:
                node_features = tf.concat([node_features, features],
axis=1)

            if self.use_bias:
                node_features = k.backend.bias_add(node_features,
self.biases[head])

            # Add output of attention head to final output
            outputs.append(node_features)

        # Aggregate the heads' output according to the reduction method
        if self.attn_heads_reduction == 'concat':
            output = k.backend.concatenate(outputs)  # (N x KF')
        else:
            output = k.backend.mean(k.backend.stack(outputs), axis=0)
# N x F')

        output = self.activation(output)
        return output

    def compute_output_shape(self, input_shape):
        output_shape = input_shape[0][0], self.output_dim
        return output_shape
```

RNN Cells

```
from
enum
import
Enum
        from tensorflow.keras.layers import LSTM as kLSTM
        from tensorflow.keras.layers import GRU as kGRU


        class CellFunction:
            """
            A switch object for using different recurrent cell types. To use,
            first create an instance, e.g.
                cell = CellFunction(CellFunction.CellType.LSTM)
            then use the variable cell as you would the resulting cell that
            would be imported from tensorflow - i.e.
                LSTM(40, return_sequences=True, kernel_regularizer=l2(self.l2_reg))
            becomes
                cell(40, return_sequences=True, kernel_regularizer=l2(self.l2_reg))
            """

            class CellType(Enum):
                LSTM, GRU = range(2)

            def __init__(self, cell_type):
                """
                :param cell_type: CellFunction.CellType Enumeration Value.
                """
                self._type = cell_type

            @property
            def type(self):
                return self._type

            @type.setter
            def type(self, value):
                if not isinstance(value, CellFunction.CellType):
                    raise ValueError("Requested type not CellFuntion.CellType")

            def __call__(self, *args, **kwargs):
                if self.type == self.CellType.LSTM or self.type ==
        self.CellType.LSTM.value:
                    return kLSTM(*args, **kwargs)
                if self.type == self.CellType.GRU or self.type ==
        self.CellType.GRU.value:
                    return kGRU(*args, **kwargs)
```

```
        raise TypeError("Unknown Cell Type. Class implementation requires
updating.")

    def __str__(self):
        if self.type == self.CellType.LSTM:
            return "CellFunction: LSTM"
        if self.type == self.CellType.GRU:
            return "CellFunction: GRU"
        raise TypeError("Unknown Cell Type. Class implementation requires
updating.")
```