3D SHAPE CLASSIFICATION

Index

Introduction

About

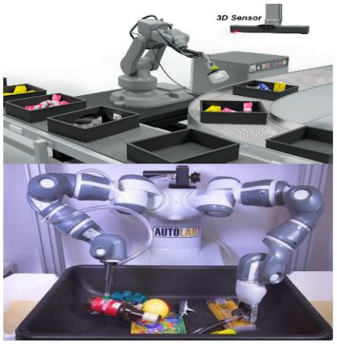
- Date:
  - 16/03/2021
- Authors:
  - Calafell Sandiumenge, Joan
  - González Peralta, Pablo Agustín
  - Ruiz Retamal, Juan Pedro
  - Vidal i Bazan, Sergi
- Advisor:
  - Mosella Montoro, Albert
- Institute:
  - Universitat Politècnica de Catalunya

Motivation

During the last few years, 3D image capture systems have started to become popular due to applications such as capturing the environment in autonomous driving or robotics. This type of application due to its complexity to, for example, recognize objects requires analysis systems based on artificial intelligence. In addition, some of the architectures used to solve these classification problems can be used for other fields such as the study of computational fluid dynamics. For more detailed information, the reader is referred to a complete review on methods and applications for Graph Neural Networks published by Zhou et al. in 2018 [ZHO18].

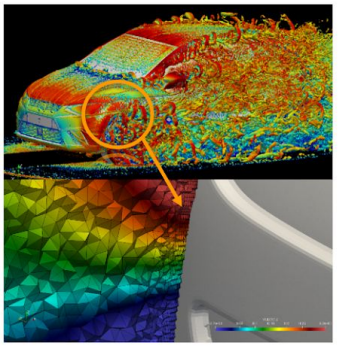
The following are some examples of 3D Shape Classification applications:

ROBOTICS
e.g. Bin Picking Applications
Automatic organization by class of mixed objects



AUTONOMOUS DRIVING
Classification & Segmentation of objects in point clouds generated by LIDAR

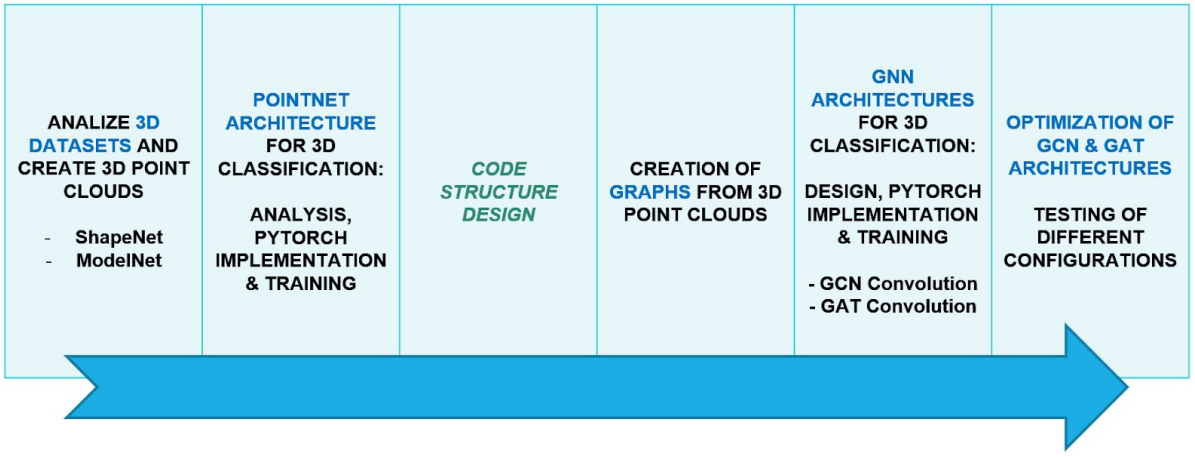
SCIENTIFIC COMPUTING
e.g. Computational Fluid Dynamics over a spatial mesh



Project Objectives

- Reproduce from scratch network architectures described in scientific publications.
- Learn to work with 3D point clouds and use them in deep learning.
- Work with Point Clouds and be able to derive Graphs from them.
- Understand how to apply this knowledge to Computer Vision and other applications like Fluid Dynamics Simulations.

Project Scope and Timeline



Milestones

- Implementation of PointNet for object classification, one of the first NN that directly consumes Point Clouds, (DOI: 10.1109/CVPR.2017.16) with Pytorch and Pytorch Geometric.
- Training and optimization of neuronal networks with ModelNet dataset.
- Build Graphs from Point Clouds.
- Implement GNN to improve the baseline performance.
- Training, optimization and testing the GNN.

ModelNet Dataset

The ModelNet dataset contains a collection of CAD models of household objects such as chairs or beds. In this project, we have worked with ModelNet10 which includes 4899

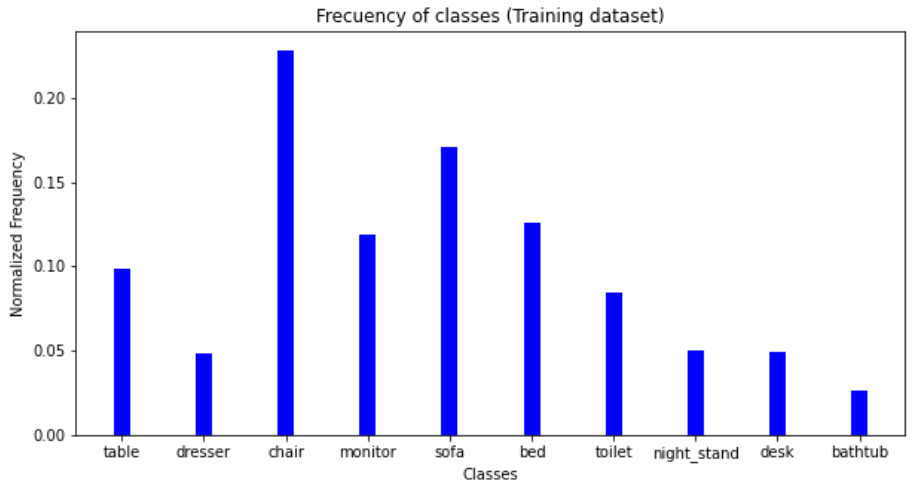
CAD models divided into 10 different categories of objects. These elements are also separated into the categories of 'Train' and 'Test'. To obtain the validation dataset separately, 20% of the elements are taken randomly from the training dataset. To train, validate and test a point cloud has been generated from these CAD models. This point cloud is composed in each model of 1024 points generated randomly on the surface of the CAD model.

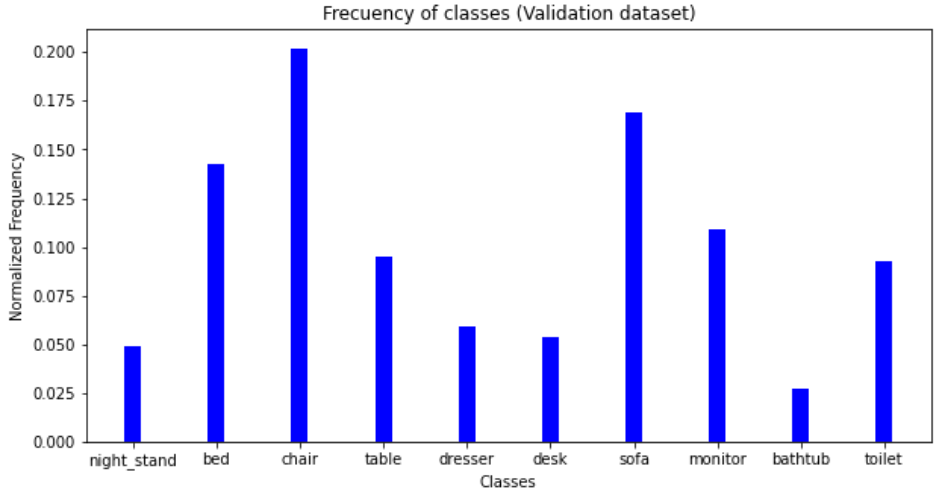
To guarantee the reproducibility of the experiments, the random indexes of elements of the dataset have been saved in txt files. These are loaded each time to take the elements in the same way always.

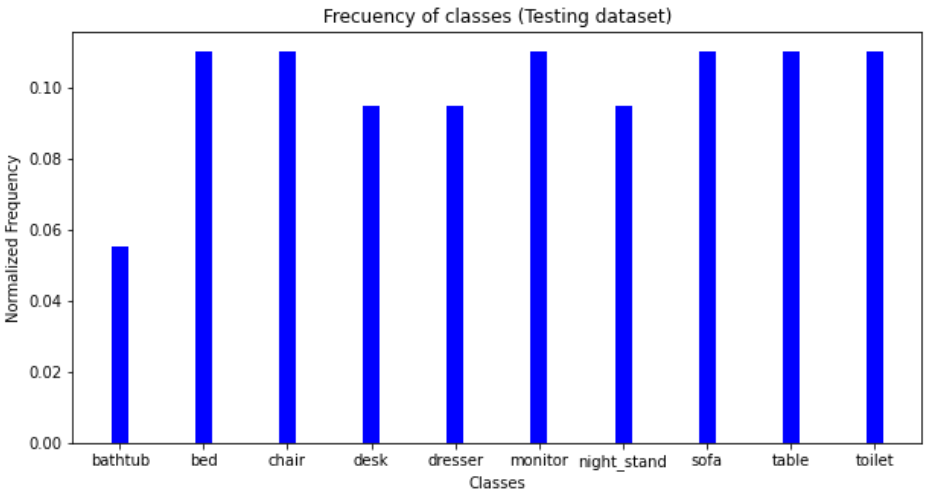
The following number of point clouds and labels were used for each split:

- Test: 3193
- Validation: 798
- Train: 908

Ideally, within the dataset, each of the classes would have the same number of elements as the rest of the classes, but as we can see in these histograms, the frequency of appearance of each class is very different.







From Mesh to Point Clouds to Graphs

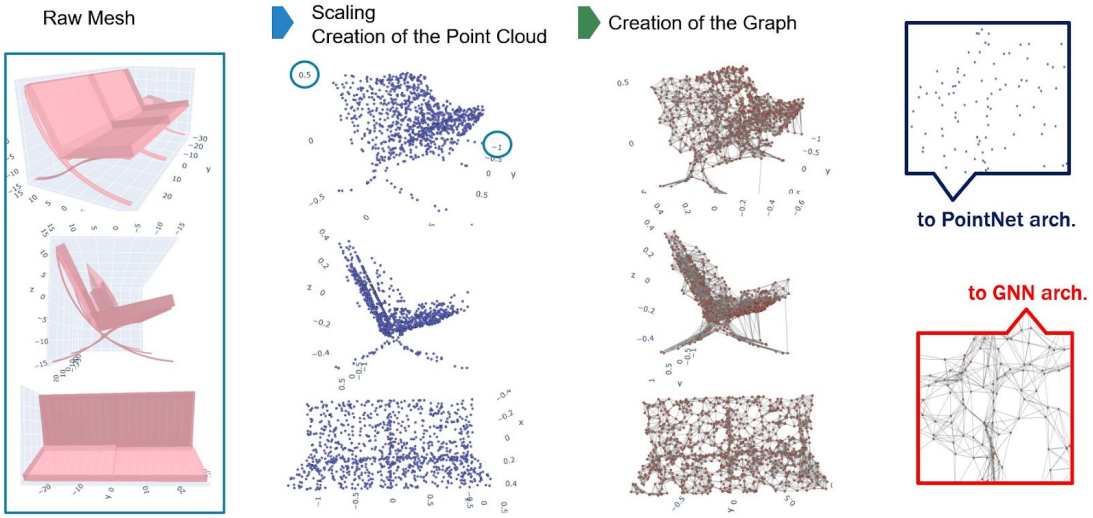
The Modelnet dataset consists of meshes defined by faces and vertices. For our experiments with point clouds and graphs networks, we need first to create them from the meshes.

We use pytorch geometric to sample a fixed number of points (3D) from the faces of the object. The obtained point cloud can be then fed into the PointNet architecture.

When using Graph Neural Networks, we need to convert the 3D Point Cloud into a Graph with nodes and edges. The nodes will correspond to the points sampled in the previous operation. Each node will contain a 3-dimensional feature vector containing its x, y and z coordinates in the point cloud. The edges will be found using a KNN algorithm that, for each node, it will find its closest '9' nodes and create the edges from the source node towards them.

Note that the number of neighbours '9' has been arbitrarily chosen for our experiments in this project and will be used in all the experiments below.

As described above, the figure below summarizes the operations performed to prepare the dataset to feed our networks.

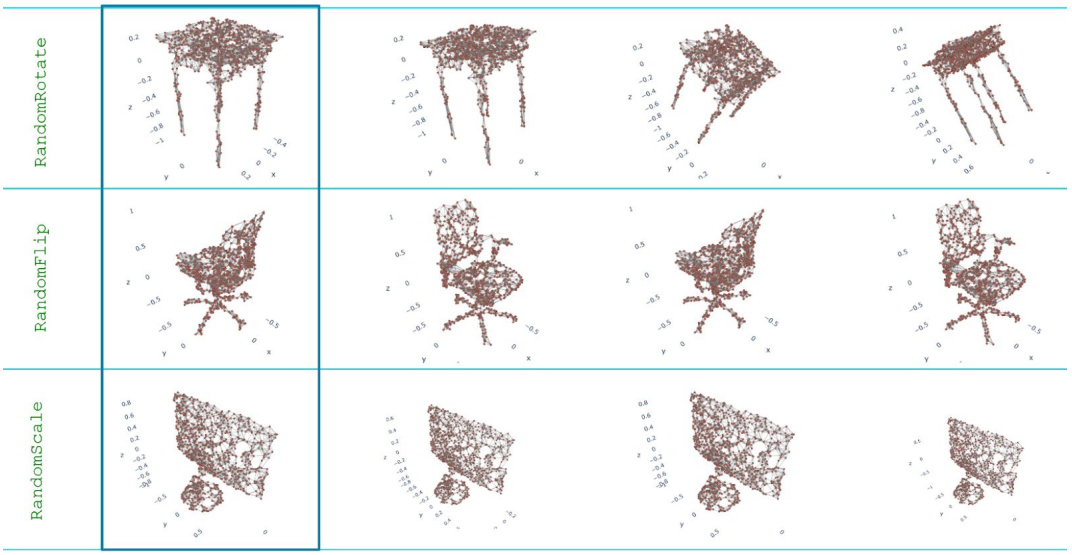


Data Augmentation

Several data augmentation techniques have been explored in the case we observe overfitting during our tests.

We have chosen the online data augmentation approach where each epoch will see a different dataset where the point clouds are randomly transformed when called into the batch. This approach allows to save space in the drive and uses the same amount of samples (size of the dataset fed into the train loop) in each test (with or without augmentation). The drawback is that the retrieval of the data is slower as a transformation is done ‘online’ before adding the image into the batch.

Three transformations from torch_geometric.transforms have been retained: RandomRotate, RandomFlip and RandomScale. The effects on the graphs of these transformations is illustrated below. The image highlighted in the square is the original one.

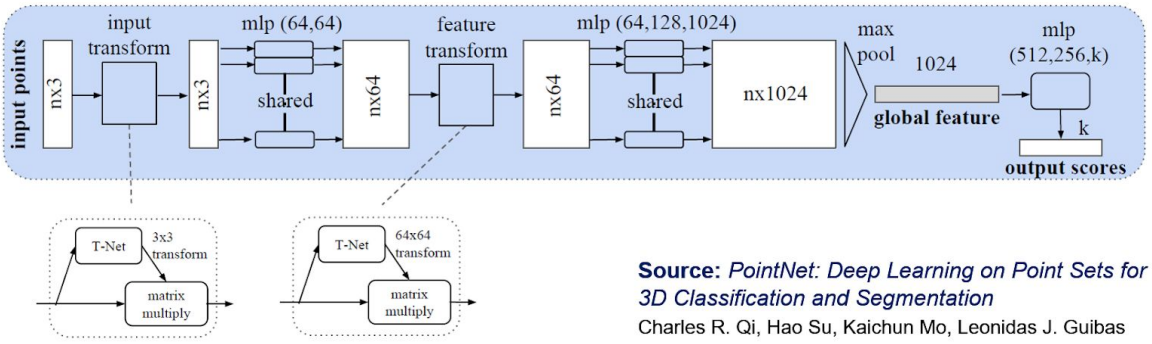


Neuronal Network Architectures

PointNet

PointNet was the first deep learning framework to directly handle 3D point clouds. It provides a unified architecture for applications ranging from object classification, part segmentation, to scene semantic parsing.

As our objective is focused on classification task, we have only considered the Classification network part, shown below:



Source: *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation*
Charles R. Qi, Hao Su, Kaichun Mo, Leonidas J. Guibas

The classification network takes N points as input, applies input and feature transformations, and then aggregates point features by a max pooling layer. The output is a classification score for k classes.

Note that the number of trainable parameters of the network is 3,463,763.

The network has three main parts:

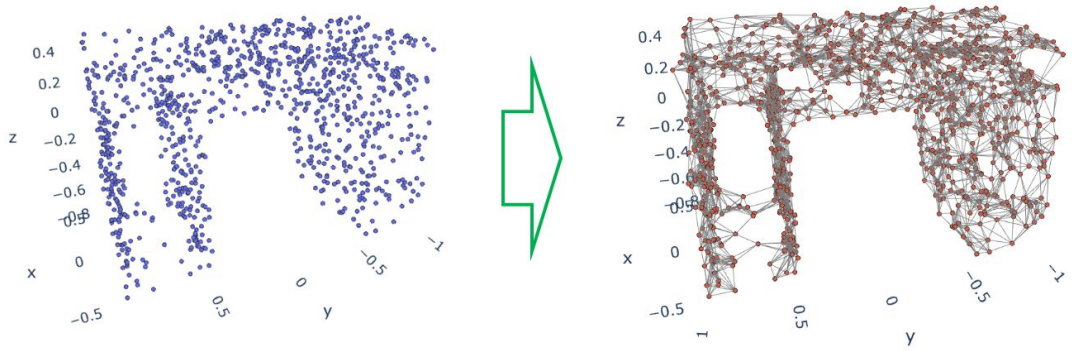
The Input and feature transform are two transformation networks (called T-net) that align input points and point features to make them invariant to certain transformations such as rotation or translation of the whole point set. (T-net is a mini PointNet composed by a shared MLP, a max pooling and two fully connected layers).

The shared MLPs are used for learning a spatial and feature encoding for each point and mapping them to a higher-dimensional space.

The max pooling layer is used to aggregate information from all the points, and gives as output a global feature vector that is invariant to the input order.

Graph Neuronal Networks

By transforming Point Clouds into Graphs, we have the possibility of using Graph Convolutions and creating neural networks for 3D classification based on graphs.



Two specific convolutions have been applied in our project. They are described in the following publications and implemented in Pytorch Geometric

- **Graph Convolutional Networks** as described in “Semi-Supervised Classification with Graph Convolutional Networks” [KIP16]
- **Graph Attention Networks** as described in “Graph Attention Networks” [VEL18]

Below, we briefly describe both approaches and the way the output feature vectors are computed. Note that the formulas and illustrations describing the convolutions have been directly taken from the works cited above.

Graph Convolutional Networks

The output features of the convolution are calculated in the following way:

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \frac{1}{c_{ij}} W^{(l)} h_j^{(l)}\right)$$

$h^{(l)} \in \mathbb{R}^F$
$h^{(l+1)} \in \mathbb{R}^{F'}$
$W \in \mathbb{R}^{F' \times F}$

$c_{ij} = \sqrt{d_i d_j}$, where *d* is the node degree

- h_j is the current feature vector (dimension F) of a node neighbour to i (connected through an edge).
- The output feature vector of i (dimension F’) is calculated adding the result of the computation of all its neighbours features multiplied by the weight matrix. In this way the “knowledge” of its neighbours N(i) is transferred towards the node.

- The addition is normalized by cij, a constant for the edge i-j
- W is the matrix of weights to be learned.
- F and F' are the dimensions of, respectively, the input and output feature vectors of the node.
- This operation is isotropic (no direction or node is privileged over the other)
- Only the nodes of first degree influence the output of the convolution. However, by stacking several convolutions of this type, we can make the nodes of higher degrees to effectively influence the feature map of the node.

Pytorch Geometric implementation: Class **GCNConv** (link)

Graph Attention Networks

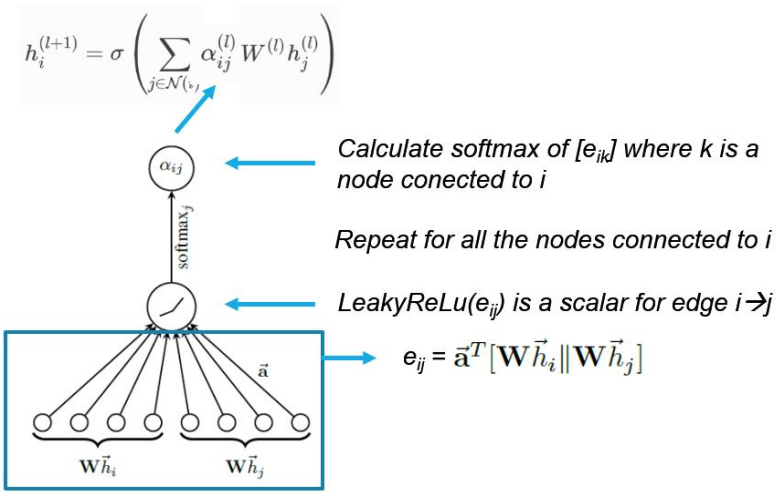
In this type of networks, the output features are calculated using an attention mechanism that, contrary to the previous approach, privileges the features of some neighbour nodes versus the others.

The calculation then multiplies the feature vectors of the neighbouring nodes to i with the weight matrix and then by an attention coefficient α. The condensed formula is below:

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} W^{(l)} h_j^{(l)}\right) \qquad \begin{array}{l} h^{(l)} \in \mathbb{R}^F \\ h^{(l+1)} \in \mathbb{R}^{F'} \\ W \in \mathbb{R}^{F' \times F} \end{array}$$

- F and F' are the dimensions of, respectively, the input and output feature vectors of the node
- W is the matrix of weights to be learned
- α is computed through a softmax so the output is normalized
- Nodes are able to "attend" over their neighborhoods' features giving different weights to different nodes in a neighborhood

The following figure describes how the calculation of the attention coefficient α is done (read from the bottom to the top).



- h_i and h_j are the current feature vectors of nodes i and j
- j is a node neighbour of node i
- $\vec{a}$ is a vector learned by the network

a, vector of dimension 2xF' (with values learned by the network), is a shared attentional mechanism used to compute the attention coefficients.

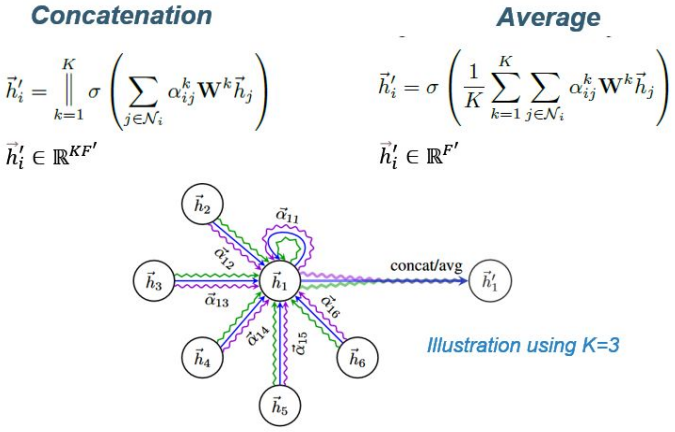
GATs can have multiple attention heads. Each attention head computes a separate output feature vector $h_i^{(l+1)}$.

The output of the convolution can then combine those vectors in 2 different ways: concatenation or average.

If concatenation is used, each one of the output feature vectors of the node (there will be one for each one of the heads), is concatenated to each other resulting in a vector of dimension K * F (being K the number of attention heads).

If average is used, the output feature vectors are averaged and the dimensionality of the output is independent of the number of attention heads used.

The following figure (taken from [VEL18]) illustrate the process:



Pytorch Geometric implementation: Class **GATConv** (link)

Tests setup and results

In this section, the design of the testing campaign will be detailed for each architecture implemented in the project. The evolution of the network architectures has followed a logical sequence, starting from an initial baseline and making successive improvements that have been decided on the basis of the previous test results. Along this section, the different tested configurations and their justification will be presented together with their respective results. At the end of each architecture subsection, a table summarizing all the testing campaigns will be shown.

The criteria for the baseline selection have differed for each network architecture. The PointNet experiment was intended to learn how to implement a full network from scratch by reading and understanding the original article. For that reason, the experiment baseline was the paper's original architecture. The two other experiments were intended to reduce the size of the network in terms of complexity and trainable parameters. To that end, simple convolutional architectures featuring the minimum and essential set of layers and enhancing techniques have been used as a baseline. For essential layers and techniques we mean those methodologies that have proven their general ability in improving training results and providing training efficiency and stability such as input and batch normalization.

PointNet Experiment

The primary motivation of this experiment is to learn how to implement a functional neural network from scratch by reading and understanding the original article of Qi et al. [QIC16]. Additionally, testing our implementation by comparing its classification performance against existing benchmarks [GAR16] is another fundamental target of this first experiment. A complete list of the classification accuracy of different network architectures, including PointNet for the Modelnet10 and Modelnet40 datasets can be found in the following link: https://modelnet.cs.princeton.edu/

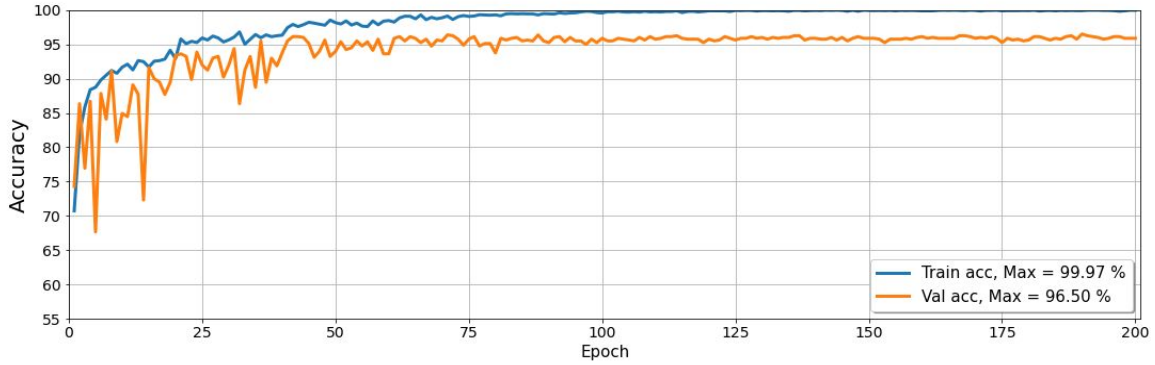
Experiment setup:

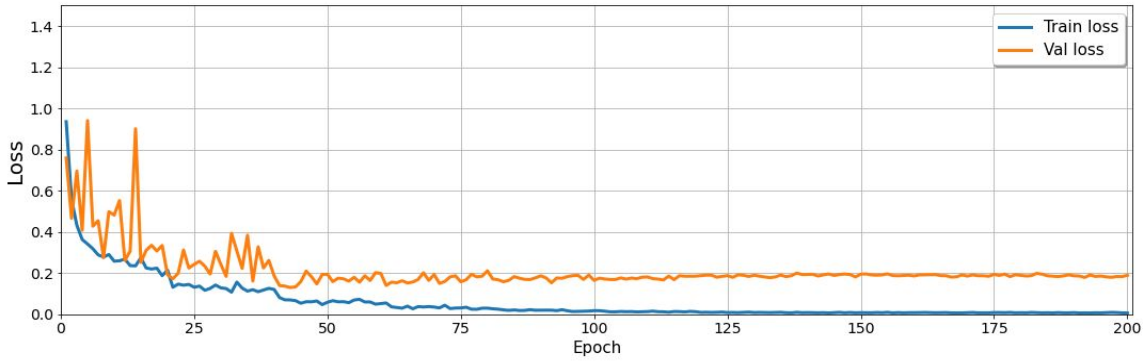
The experiment is intended to replicate the original Pointnet architecture, and thus, we seek to achieve similar results to those obtained by the original paper's authors. As commented in the Architectures section, PointNet is mainly based on MLPs. This approach entails a large number of trainable parameters which is a major drawback due to the required computational and memory resources. Moreover, the architecture also features a secondary net called T-net which is responsible for making the overall architecture insensitive to geometry displacements and rotations and to the input sequence order. Additionally, batch normalization layers as well as input normalization are included to provide training stability and improve accuracy for both, the training and the validation datasets. Finally, a max pooling layer is added in order to reduce the dimensionality prior to the MLP classification step. Regarding the data, the Modelnet10 dataset has been used for all tested configurations.

Experiment results and conclusions:

Architecture	Max. Train Acc. (%)	Max. Val. Acc. (%)
Test 1: Baseline	99,97	96,5

The obtained results in validation accuracy (96.5%) clearly outperform the scores obtained by the reference paper that tested ModelNet10 with PointNet [GAR16], reaching a value of 77.6%. In plots below, the training and validation accuracy curves together with their loss counterparts are displayed. While the training accuracy curve seems pretty stable, the validation one shows an unstable behavior at the beginning, featuring large peaks. On the other hand, thanks to overfitting preventing measures such as Dropout and Weight Decay, the validation and the test loss and accuracy curves converge into parallel evolutions. Moreover, the gap between train and validation accuracies is less than 5% which indicates a reasonable performance regarding the applied overfitting prevention measures.





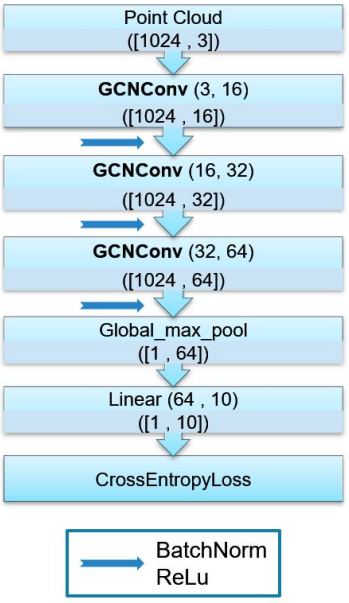
	Performance Improvement			Reduce Overfitting		Metrics for Decision Making	
Test	Input Scaling	Batch Norm	l.r. scheduler	Dropout	Weight decay	Max. **Train** Accuracy	Max. **Val.** Accuracy
PointNet	(-1,1)	TRUE	0.001	p=0.3	0.001	100.0%	96.5%

GNN Experiments

The motivation of the use of Graph Neural Networks is to allow the application of convolutional layers to drastically reduce the number of trainable parameters with respect to the architecture based on MLP while keeping the global network performance. In that sense, the experiments will be started with an initial baseline architecture designed with the primary objective of minimizing its complexity and number of trainable parameters. Depending on the test outputs, subsequent layers and methods will be introduced in order to achieve the best trade off between validation accuracy, network size, and overfitting minimization. Concerning the data, the ModelNet10 dataset is used throughout the testing campaign.

GCN

Initially, the graph convolution operator (GCNconv) of Kipf and Welling [KIP16] will be used to create the baseline of the first graph convolution network. The network architecture consists of three GCNconv layers with batch normalization and ReLu activations on top of each layer. The initial number of convolutional layers is taken from the original article [KIP16].



The first convolutional layer input consists of three node features corresponding to the spatial coordinates of the graph node. This information is essential for the network to understand the data spatial structure. Then, this first layer outputs 16 features. Accordingly, in the two subsequent convolutional layers, the input number of features are 16 and 32, and their output sizes are 32 and 64 respectively.

After the convolutional block, a fully connected layer with an output size equal to the number of the dataset categories is placed for classification purposes. Moreover, between the convolutional block and the MLP, a pooling layer is applied to aggregate information from all the points, and gives as output a global feature vector that is invariant to the input order.

Test 1: Baseline. Average vs. Max Pooling

Experiment setup:

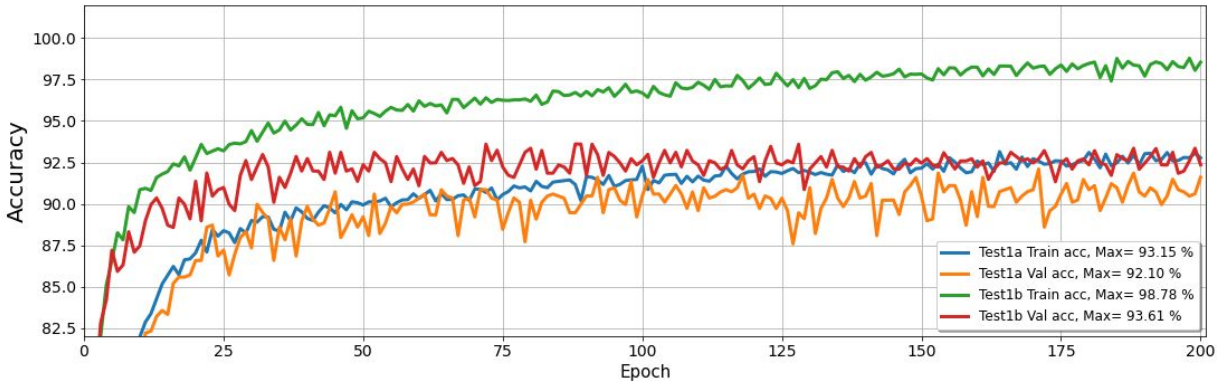
This first test is intended to determine which pooling layer is more suitable for the present architecture. To do so, two different tests have been carried out by training the baseline architecture featuring two different pooling layers, a max and an average pool respectively. For the present test, a constant learning rate of 0.001 has been used for both experiments. The architecture setup is summarized hereafter:

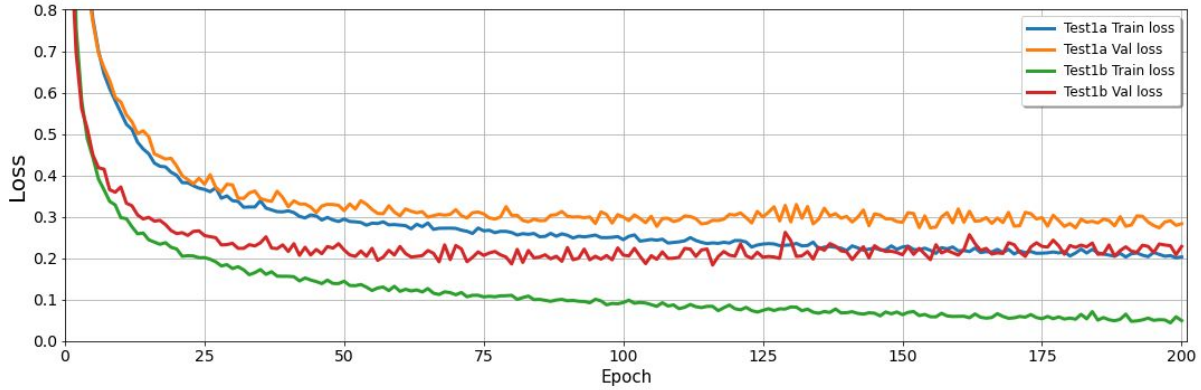
- Test 1a = 3 conv. layers - (3, 16)(16, 32)(32, 64) + avg pool + MLP(64,10)
- Test 1b = 3 conv. layers - (3, 16)(16, 32)(32, 64) + max pool + MLP(64,10)

Experiment results and conclusions:

Architecture	Max. Train Acc. (%)	Max. Val. Acc. (%)
Test 1a: Baseline w/ avg pool	93.15	92.10
Test 1b: Baseline w/ max pool	98.78	93.61

The experiment results show clearly that the max pool approach performs significantly better than the average pool either for the training and the validation accuracy. Therefore, the max pool will be applied from this experiment on as the default pooling layer for the baseline. Nonetheless, the gap between training and validation accuracy and loss curves is significantly larger for the max pool layer, showing signs of overfitting that will have to be addressed on the networks development final stages.





Test 2: Baseline. Learning rate effects

In this second test, we want to investigate the effects of modifying the learning rate (lr) value. To that end, the best performing architecture in the previous experiment, the baseline featuring max pooling, will be used for testing.

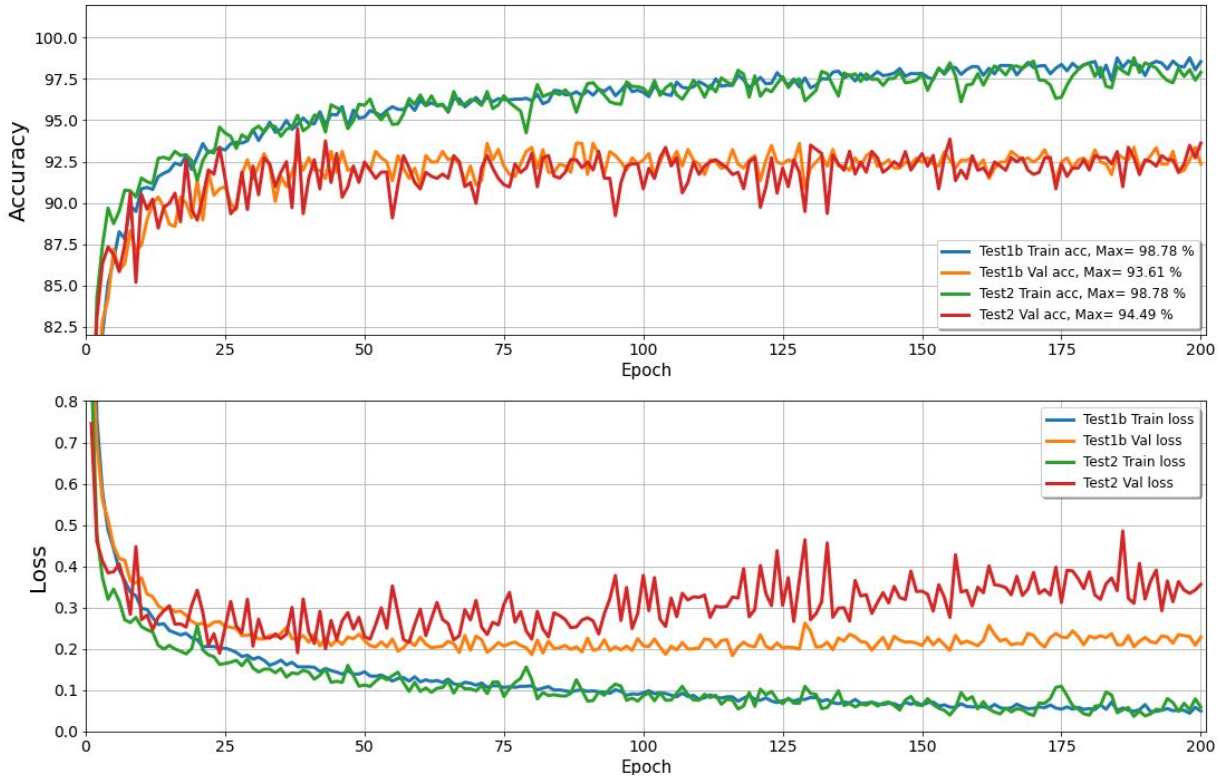
Experiment setup:

The baseline architecture with max pooling will be used for the test. In Test 1b, the model was trained with a lr=0.001. In the present experiment (Test 2), the learning rate value will be increased by one order of magnitude to determine the effects of using a larger lr value on convergence stability as well as on training and validation accuracies.

- Test 1b = 3 conv layers-(3, 16)(16, 32)(32, 64) + max pool + MLP(64,10) (lr=0.001)
- Test 2 = 3 conv. layers-(3, 16)(16, 32)(32, 64) + max pool + MLP(64,10) (lr=0.01)

Experiment results and conclusions:

Architecture	Max. Train Acc. (%)	Max. Val. Acc. (%)
Test 1b: Baseline lr = 0.001	98.78	93.61
Test 2 Baseline lr = 0.01	98.78	94.49



Increasing the learning rate value from 0.001 to 0.01 has produced a slight increase in the validation accuracy. For that reason, the later value will be used from this test on. However, this accuracy improvement has come at expenses of training stability. Either the validation loss and accuracy curves show an increased fluctuating component with larger peaks which seems a logical outcome of using a larger learning rate value. To prevent this undesired effect, a learning rate scheduler method will be used in the next test. Moreover, a significant increase in the overfitting is observed since the gap between training and validation loss

curves has significantly widened. Nevertheless, as in the previous experiment, this drawback will be dealt with at the end of the testing campaign.

Test 3: Baseline with learning rate scheduler.

This experiment is intended to smooth the validation and training curves either for the accuracy and losses. The increase of the learning rate value has improved the validation accuracy at expenses of the training stability, and with this experiment we seek to keep a high value of classification accuracy while having a learning process as stable as possible.

Experiment setup:

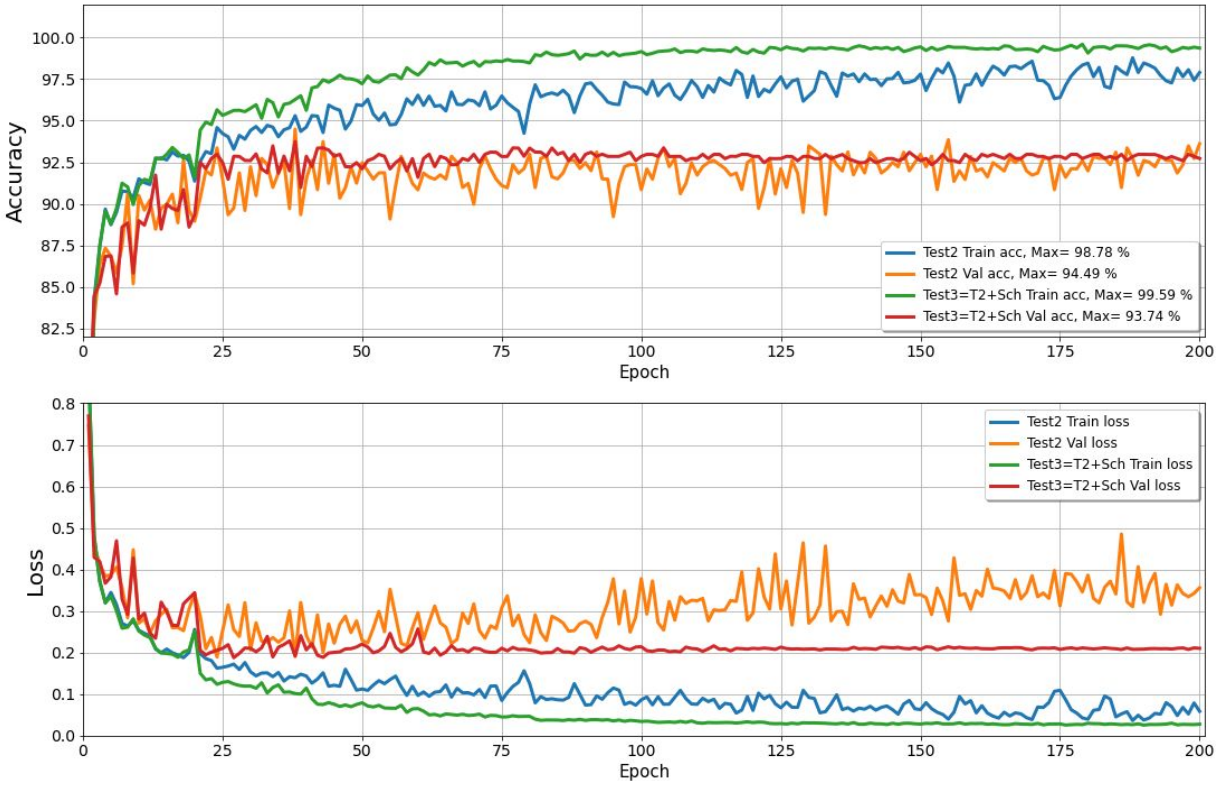
Test 2 will be used as a baseline for this test. The only change will be the inclusion of a learning rate scheduler in the training function intended to increase the learning process stability.

- Test 2 = 3 conv. layers - (3, 16)(16, 32)(32, 64) + max pool + MLP(64,10) (lr=0.01)
- Test 3 = Test 2 + learning rate scheduler

Experiment results and conclusions:

Architecture	**Max. Train Acc. (%)**	**Max. Val. Acc. (%)**
Test 2 Baseline lr = 0.01	98.78	94.49
Test 3 Baseline lr scheduler	99.59	93.74

The learning rate scheduled has reversed the effects of using a larger learning rate value. While the training curves have been significantly smoothed and the overfitting has sharply dropped, the validation accuracy has slightly decreased. We think that the gains of using a scheduler clearly outweighs the minor reduction of the validation accuracy. Nonetheless, we will use other methods such as increasing the convolutional layer sizes to improve the current validation accuracy scores.



Test 4: Baseline with learning rate scheduler and double layer capacity.

At this point, the train accuracy stands at 99.59% while the learning process shows a pretty smooth and stable behavior. This fourth experiment is aimed at increasing validation accuracy and trying to close the gap between the test and the validation accuracy curves. To that end, the number of features generated by each convolutional layer will be doubled, and their effects will be subsequently analyzed.

Experiment setup:

In the present experiment, the baseline architecture with learning rate scheduler will be modified in order to increase the number of features generated by convolutional layers. Below, details of the architecture modifications are given:

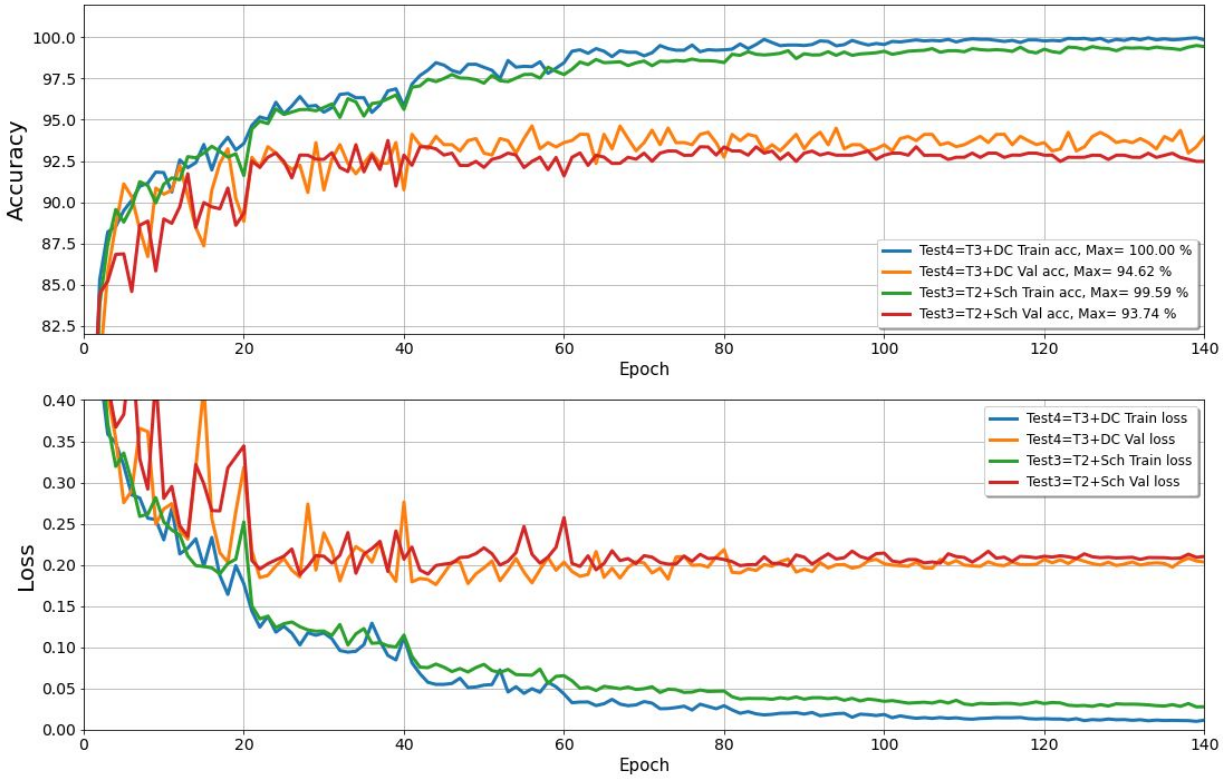
- Test 3=3 conv. layers-(3, 16)(16, 32)(32, 64) + max pool + MLP(64,10)
- Test 4=3 conv. layers-(3, 32)(32, 64)(64, 128) + max pool + MLP(128,10)

Experiment results and conclusions:

Architecture	**Max. Train Acc. (%)**	**Max. Val. Acc. (%)**
Test 3 Baseline lr scheduler	99.59	93.74
Test 4 Baseline lr scheduler double capacity	100	94.63

The increase of the convolutional layer capacity in terms of number of features has slightly increased both, the training and the validation accuracy. In fact, the training accuracy

reached 100%, showing the ability of the network to fully learn the training dataset. However, an even higher increase in the validation accuracy has been obtained, resulting in a smaller gap between both values. On the other hand, the capacity enlargement has caused slight increase of the overfitting due to the significant reduction of the training losses which widened the gap with the validation ones. For that reason, in the following experiments, the overfitting problem will be adequately addressed by applying regularization techniques.



Regularization

In this final step, regularization techniques will be applied to the previous architecture, which is the best performing one so far. The different measures applied to improve either the training and validation accuracies have caused some overfitting effects that will be corrected through standard regularization techniques. Those measures include dropout, and data augmentation generated with random rotations and flips.

Experiment setup:

In this final set of tests, regularization techniques will be applied separately to the best performing architecture (Test 4). These techniques include dropout, and data augmentation through random flips and rotations:

- Test 5 = Test 4 + Dropout
- Test 6 = Test 4 + Random Flip
- Test 7 = Test 4 + Random Rotate

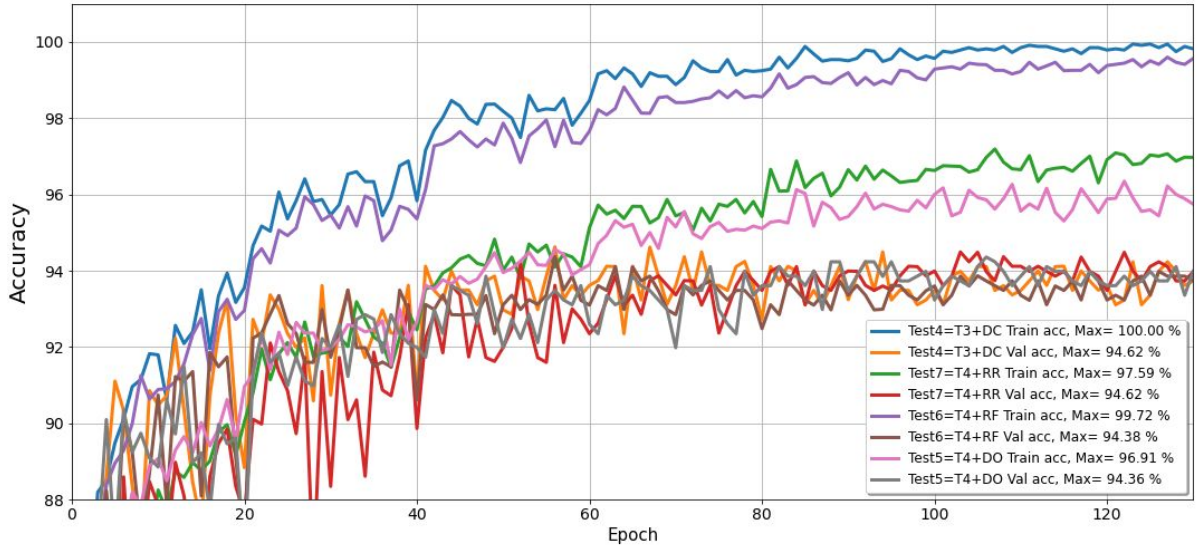
Experiment results and conclusions:

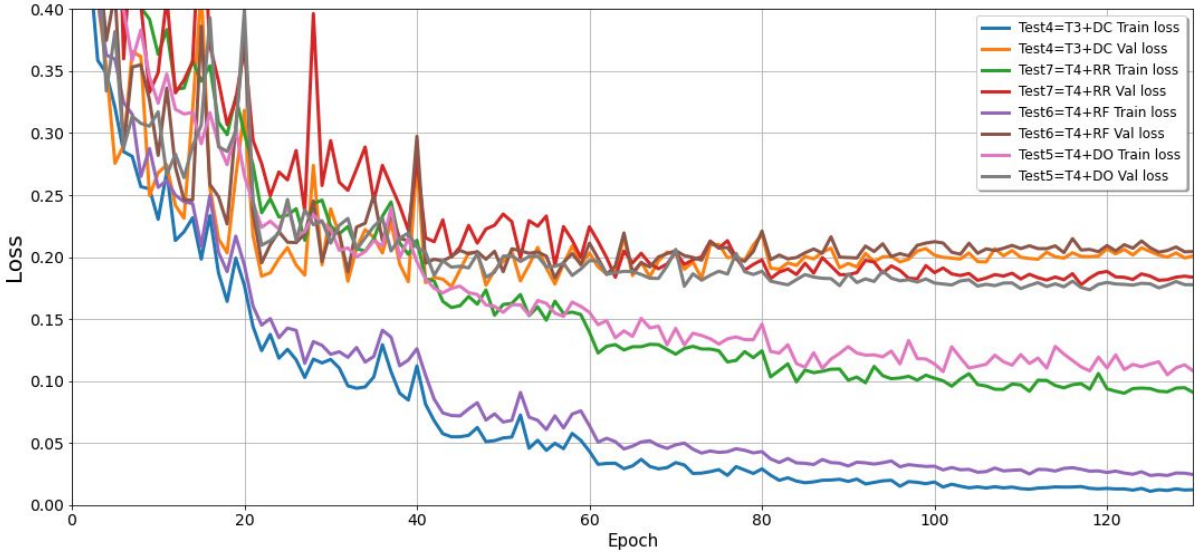
Although the purpose of the present set of experiments is not to improve the accuracy values but reducing the overfitting, a summary of training and validation accuracy values is

presented below to analyze the effects of the different regularization techniques on these magnitudes.

Architecture	Max. Train Acc. (%)	Max. Val. Acc. (%)
Test 4	100	94.63
Test 4 + Dropout	96.91	94.36
Test 4 + Random Flip	99.72	94.38
Test 4 + Random Rotate	97.59	94.62

According to the plots below, either the dropout and the random rotate transform are quite successful in reducing the overfitting. On the other hand, it seems that the random flip rotate has little effect on this aspect. However, the dropout technique has a significant negative impact on the training accuracy. For that reason, it makes the random rotate transform the regularization technique with the best trade off between train and validation accuracies and overfitting minimization.



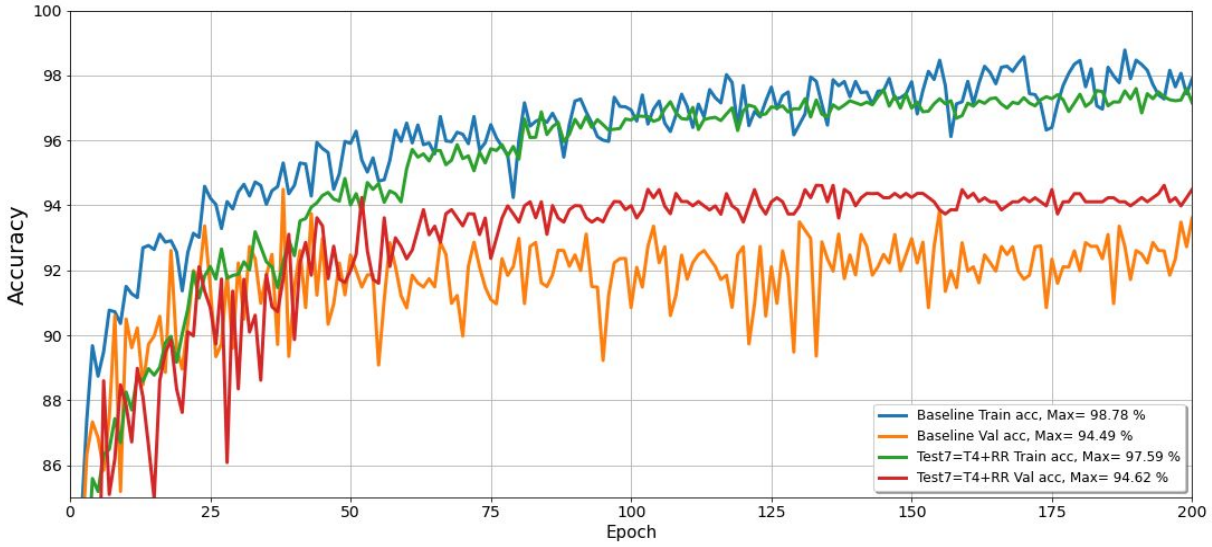


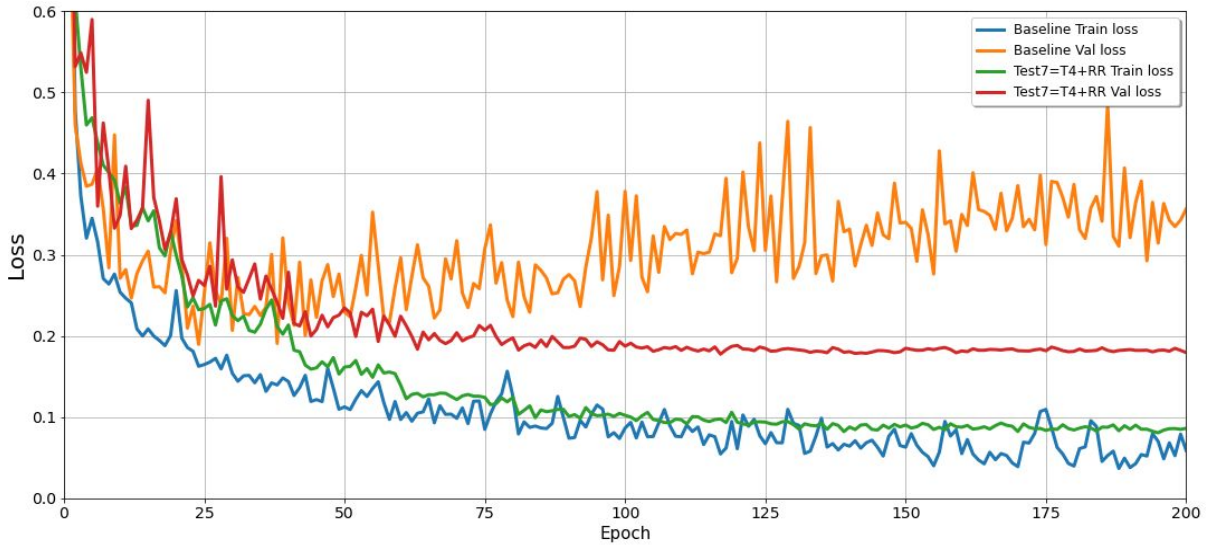
Final Conclusions (GCN Tests)

Testing campaign summary table

Test	Dimensionality Reduction		Performance Improvement		Reduce Overfitting		Metrics for Decision Making		# Trainable Weights
	Avg pool	Max pool	Scheduler	Double Capacity	Dropout	Data Augment.	Max. **Train** Accuracy	Max. **Val.** Accuracy	
T1a	lr 1e-3						93.2%	92.1%	3.594
T1b		lr 1e-3					98.8%	93.6%	3.594
T2		lr 1e-2					98.8%	94.5%	3.594
T3			lr 1e-2				99.6%	93.7%	3.594
T4			lr 1e-2				100.0%	94.6%	12.298
T5			lr 1e-2		p=0.2		96.9%	94.4%	12.298
T6			lr 1e-2			R.Flip	99.7%	94.4%	12.298
T7			lr 1e-2			R.Rotate	97.6%	94.6%	12.298

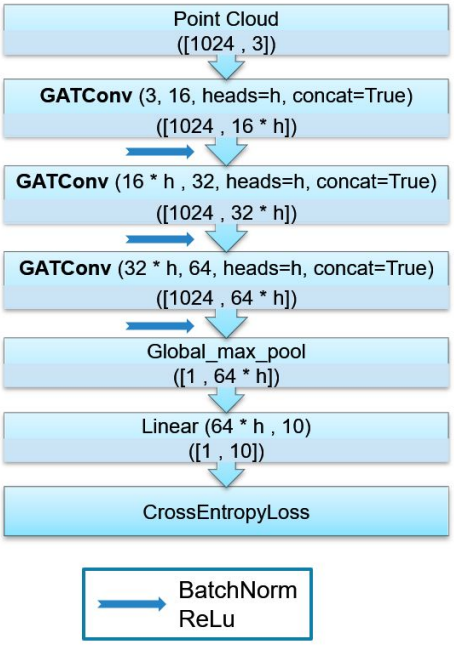
After completing the experiment campaign, we found that the best network architecture features a max pool layer together with a learning rate scheduler and increased number of convolutional features. This combination provides a high training and validation accuracy score. However, since the high capacity convolutional layers produce some overfitting, regularization techniques are also applied. After testing some different regularization approaches, we conclude that the random rotate transform significantly reduces the overfitting problem while keeping a reasonable training and validation accuracy. In the plot below, a comparison between the initial baseline and the final architecture performance in terms of accuracy and losses is shown. As it can be observed, although the training accuracy has slightly decreased, a significant improvement has been achieved for validation, while the gap between losses has been steeply narrowed, showing that the measures taken for reducing overfitting have been successful. On the other hand, the learning curves are significantly smoother, showing that the convergence process is much more robust and stable.


Accuracy
Epoch
Baseline Train acc, Max= 98.78 %
Baseline Val acc, Max= 94.49 %
Test7=T4+RR Train acc, Max= 97.59 %
Test7=T4+RR Val acc, Max= 94.62 %


Loss
Epoch
Baseline Train loss
Baseline Val loss
Test7=T4+RR Train loss
Test7=T4+RR Val loss

GAT

The proposed architecture consists of three GATconv layers with batch normalization and ReLu activations on top of each layer. We start again with three layers as proposed for the GCN architecture.



The first convolutional layer input consists of three features corresponding to the spatial coordinates of the graph node. This is essential for the net to understand the data spatial structure. We initially start our tests with 1 head (h=1). Then, it outputs 16 features. Accordingly, in the two subsequent convolutional layers, the input number of features are 16 and 32, and their output sizes are 32 and 64 respectively.

After the convolutional block, a fully connected layer with an output size equal to the number of the dataset categories is placed for classification purposes. Moreover, between the convolutional block and the MLP, a pooling layer is applied to reduce the dimensionality.

In subsequent tests we will be changing the number of heads (h) and testing not only the concatenation of heads but also the averaging of them (concat=False). When using the averaging option, the dimensionality of the output of the convolution is not multiplied by the number of heads.

Test 1: Baseline. Learning rate effects

Experiment setup:

The target of this first test is to determine which learning rate is the most suitable for training. For this, two different tests have been carried out, training the architecture with a scheduler with initial learning rates of 0.01 in test 1a and 0.001 in test 1b.

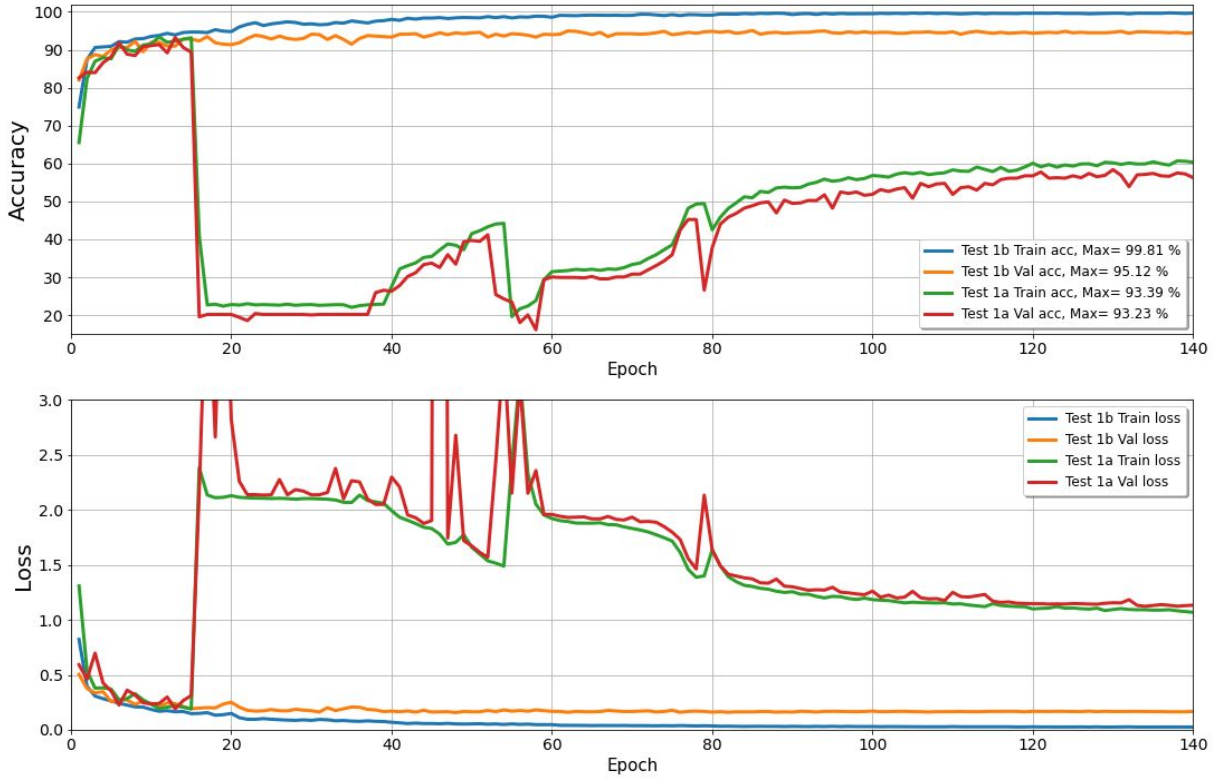
The architecture configuration is summarized below:

- Test 1a: GAT (4 Heads) 3 conv. layers-(3,16,4)(64,32,4)(128,64,4) + MLP(256,10) (init LR = 0.01)
- Test 1b: GAT (4 Heads) 3 conv. layers-(3,16,4)(64,32,4)(128,64,4) + MLP(256,10) (init LR = 0.001)

Experiment results and conclusions:

Architecture	Max. Train Acc. (%)	Max. Val. Acc. (%)
Test 1a: Baseline w/ Init LR=0.01	93.39	93.23
Test 1b: Baseline w/ Init LR=0.001	99.81	95.12

The results of this first test shows that the learning of the neural network is more efficient using an initial learning rate of 0.001. It can be seen how learning with the initial LR of 0.01 has erratic behavior from epoch 16, so maintaining this configuration would lead to problems in the following stages of network development.



Test 2: Baseline. Number of heads

Experiment setup:

In this experiment, we want to check the network's behavior if we increase the capacity to double attention heads (8 heads) or decrease to half (2 heads). These two actions decrease and increase respectively exponentially the number of network parameters, so in the first case we would have a lighter network and in the second case we would have a heavier

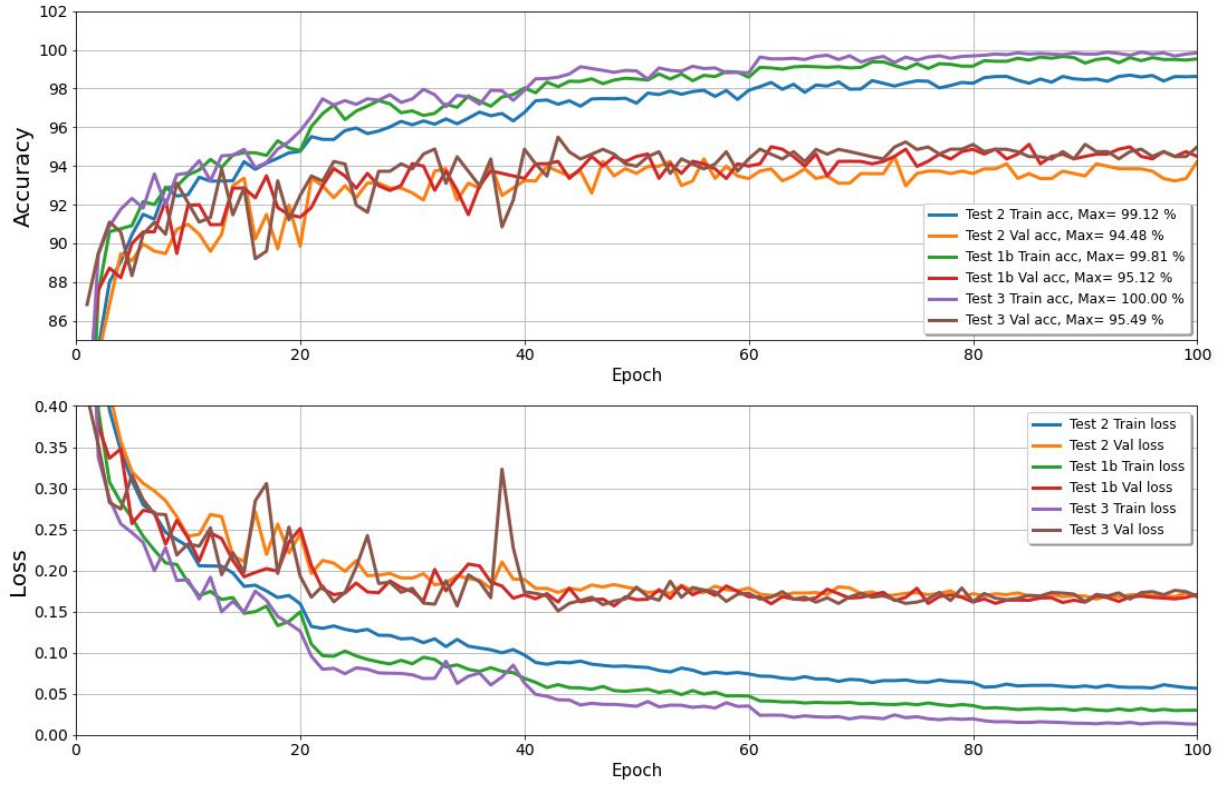
network but with more capacity to learn to identify more complex relationships and it would allow us to achieve a higher percentage of train accuracy.

- Test 1b: GAT(4 Heads) 3 conv. layers-(3,16,4)(64,32,4)(128,64,4) + MLP(256,10) (init LR = 0.001)
- Test 2: GAT(2 Heads) 3 conv. layers-(3,16,2)(32,32,2)(64,64,2) + MLP(128,10) (init LR = 0.001)
- Test 3: GAT(8 Heads) 3 conv. layers-(3,16,8)(128,32,8)(256,64,8) + MLP(512,10) (init LR = 0.001)

Experiment results and conclusions:

Architecture	Max. Train Acc. (%)	Max. Val. Acc. (%)
Test 1b: Baseline (4Heads)	99.81	95.12
Test 2: Baseline (2Heads)	99.12	94.48
Test 3: Baseline (8Heads)	100	95.49

As we can see in the graph, the results of these 3 configurations are similar. However, it can be seen that with 8 heads, the validation accuracy is on average higher, and the maximum value of train accuracy reaches 100%, which was the objective that was sought by increasing the number of heads. Regarding the 2 heads network, it can be seen that both train and validation accuracy decrease. If we continue with this network because it is lighter, we would have to increase the network's capacity as well to improve its performance. It is for these reasons that the 8-head network will be used for the following experiments.



Test 3: Baseline. Concatenate Heads vs Average heads

Experiment setup:

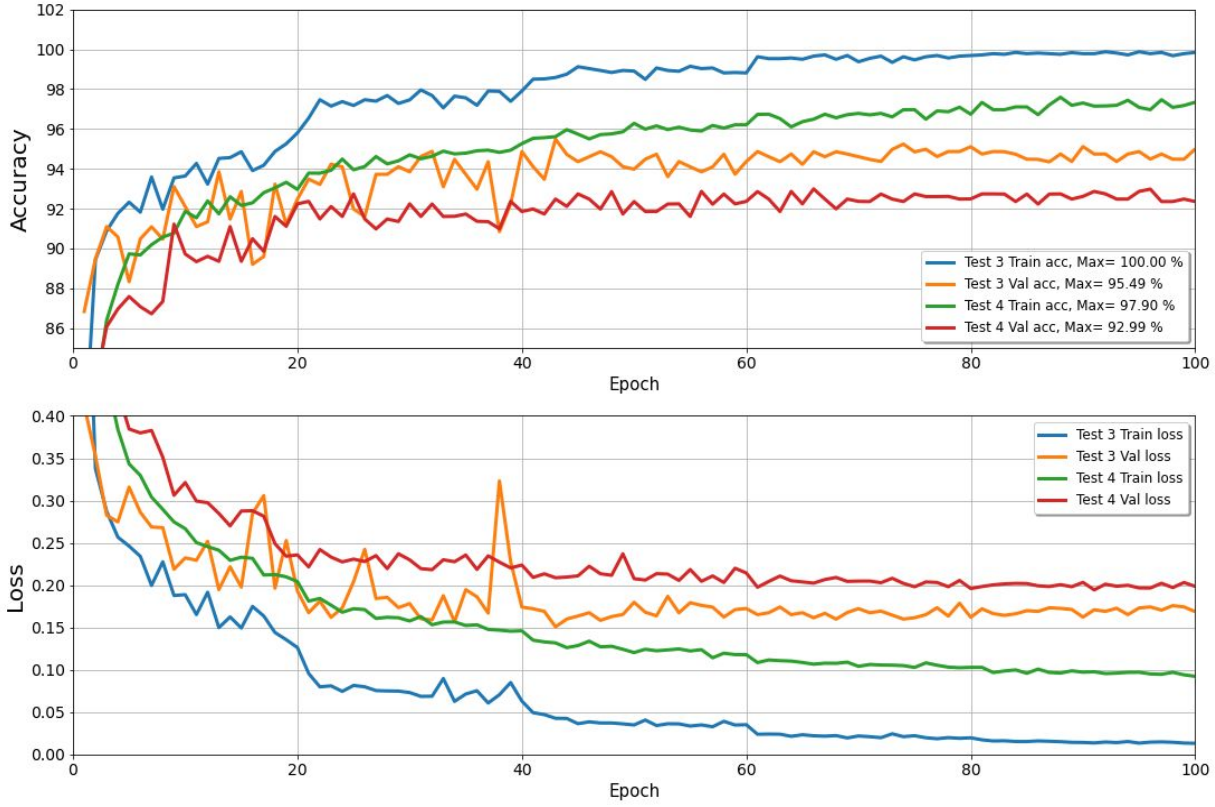
In this test, we intend to see if, by averaging all the heads in the three layers instead of concatenating them, we get a similar behavior since the network parameters are drastically reduced, but we keep the number of attention heads. The architecture of the two experiments that we are going to compare is the following

- Test 3: GAT(8 Heads) 3 conv. layers-(3,16,8)(128,32,8)(256,64,8) + MLP(512,10) (init LR = 0.001) Concatenate heads
- Test 3: GAT(8 Heads) 3 conv. layers-(3,16,8)(16,32,8)(32,64,8) + MLP(64,10) (init LR = 0.001) Average heads

Experiment results and conclusions:

Architecture	Max. Train Acc. (%)	Max. Val. Acc. (%)
Test 3: Concat. heads	100	95.49
Test 2: Average heads	97.9	92.99

As can be seen both in the graph and in the table, the performance of the architecture based on average heads, the accuracy of evaluation and train values are lower in both cases and decrease in the same proportion, so to use this architecture we would arrive to the conclusion that it is necessary to increase the capacity of the network. Therefore the best option in this case is to concatenate the heads. The next step will be to find an architecture that reduces the overfitting that can be slightly observed in the loss graph.



Regularization

In this final step, regularization techniques will be applied to the previous architecture, which is the best performing one so far. The different measures applied to improve either the training and validation accuracies have caused some overfitting effects that will be corrected through standard regularization techniques. Those measures include dropout, and data augmentation generated with random rotations and flips.

Experiment setup:

In this final set of tests, regularization techniques will be applied separately to the best performing architecture (Test 3). These techniques include dropout, and data augmentation through random flips and rotations:

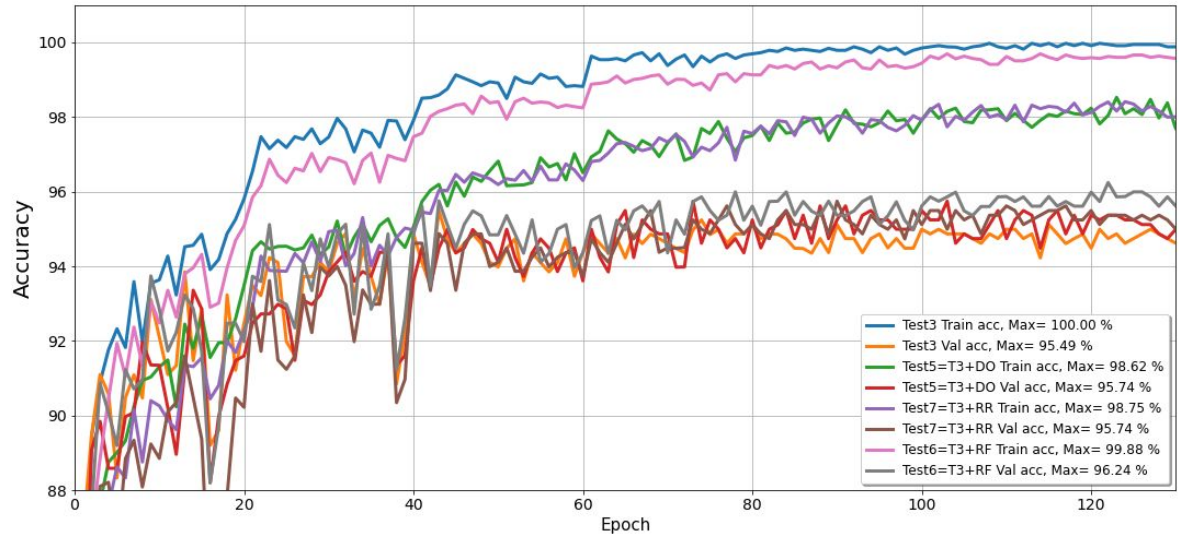
- Test 3: GAT(8 Heads) 3 conv. layers-(3,16,8)(128,32,8)(256,64,8) + MLP(512,10) (init LR = 0.001) Concatenate heads
- Test 5 = Test 3 + Dropout
- Test 6 = Test 3 + Random Flip
- Test 7 = Test 3 + Random Rotate

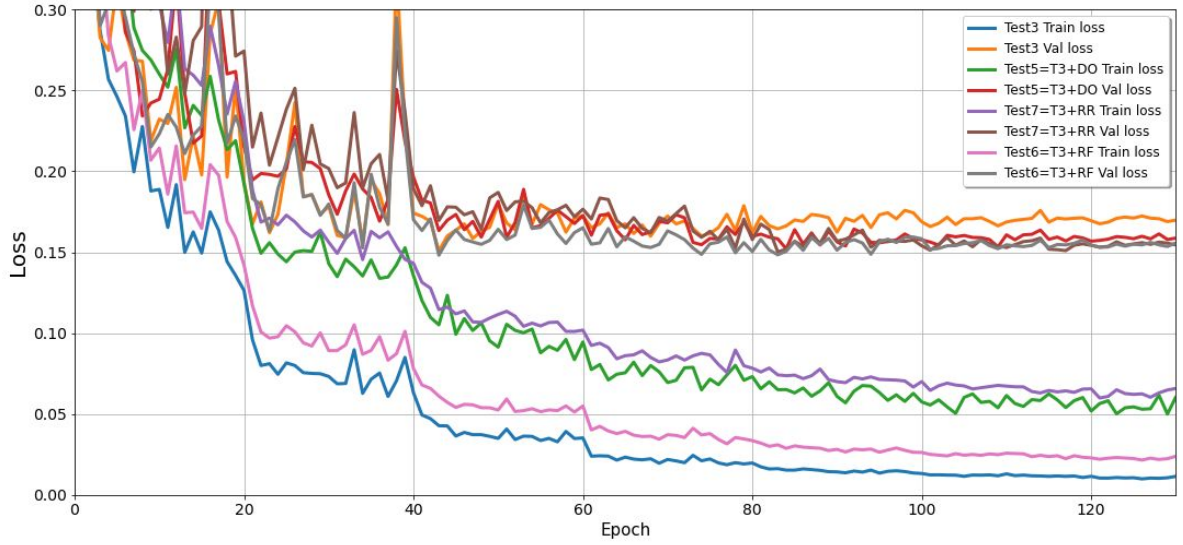
Experiment results and conclusions:

Although the purpose of the present set of experiments is not to improve the accuracy values but reducing the overfitting, a summary of training and validation accuracy values is presented below to analyze the effects of the different regularization techniques on these magnitudes.

Architecture	**Max. Train Acc. (%)**	**Max. Val. Acc. (%)**
Test 3	100	95.49
Test 3 + Dropout	98.62	95.74
Test 3 + Random Flip	99.88	96.24
Test 3 + Random Rotate	98.75	95.74

In the loss graphic, we can see how adding Dropout such as Random Flip or Random Rotate reduces overfitting while improving network performance. It can also be observed in the validation accuracy values of these three regularizations, the one with the best average and maximum performance is with Random Flip. Therefore it will be the one that we keep as the best option.



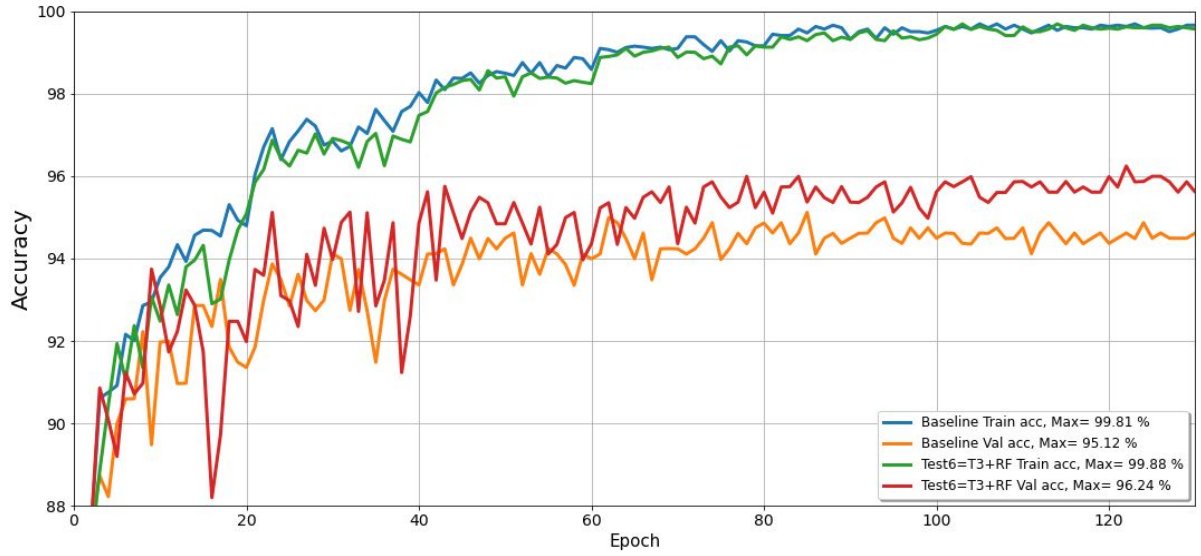


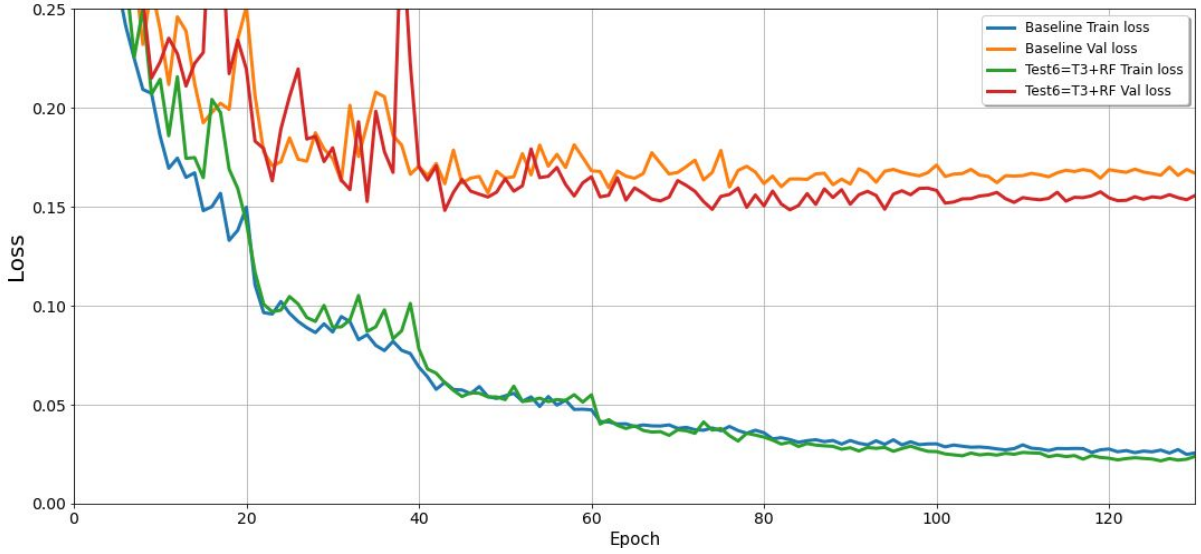
Final Conclusions (GAT Tests)

Testing campaign summary table

Test	Performance Improvement				Reduce Overfitting		Metrics for Decision Making		
	# Attention Heads	Scheduler	Concat. Att. Heads	Avg. Att. Heads	Dropout	Data Augment.	Max. **Train** Accuracy	Max. **Val.** Accuracy	# Trainable Weights
T1a	4	lr 1e-2					93.4%	92.2%	45.962
T1b	4	lr 1e-3					99.8%	95.1%	45.962
T2	2	lr 1e-3					99.1%	94.5%	12.746
T3	8	lr 1e-3					100.0%	95.5%	173.834
T4	8	lr 1e-3					97.9%	93.0%	23.642
T 5	8	lr 1e-3					98.6%	95.8%	173.834
T 6	8	lr 1e-3				R.Flip	99.9%	96.3%	173.834
T 7	8	lr 1e-3				R.Rotate	98.8%	95.7%	173.834

After completing this last phase of experiments, we observed that the best architecture is formed with 8 attention heads, a scheduler with an initial LR of 0.001, the heads' concatenation, and regularization with Random Flip as a transform. This architecture improves the accuracy of the validation. Although in the first epochs, the train accuracy is below the baseline, it ends up matching it. This final architecture manages to eliminate overfitting thanks to the regularization with RandomFlip and reach a stable result thanks to the scheduler. The graph below shows a comparison between the initial baseline and the final performance of the architecture in terms of precision and losses:

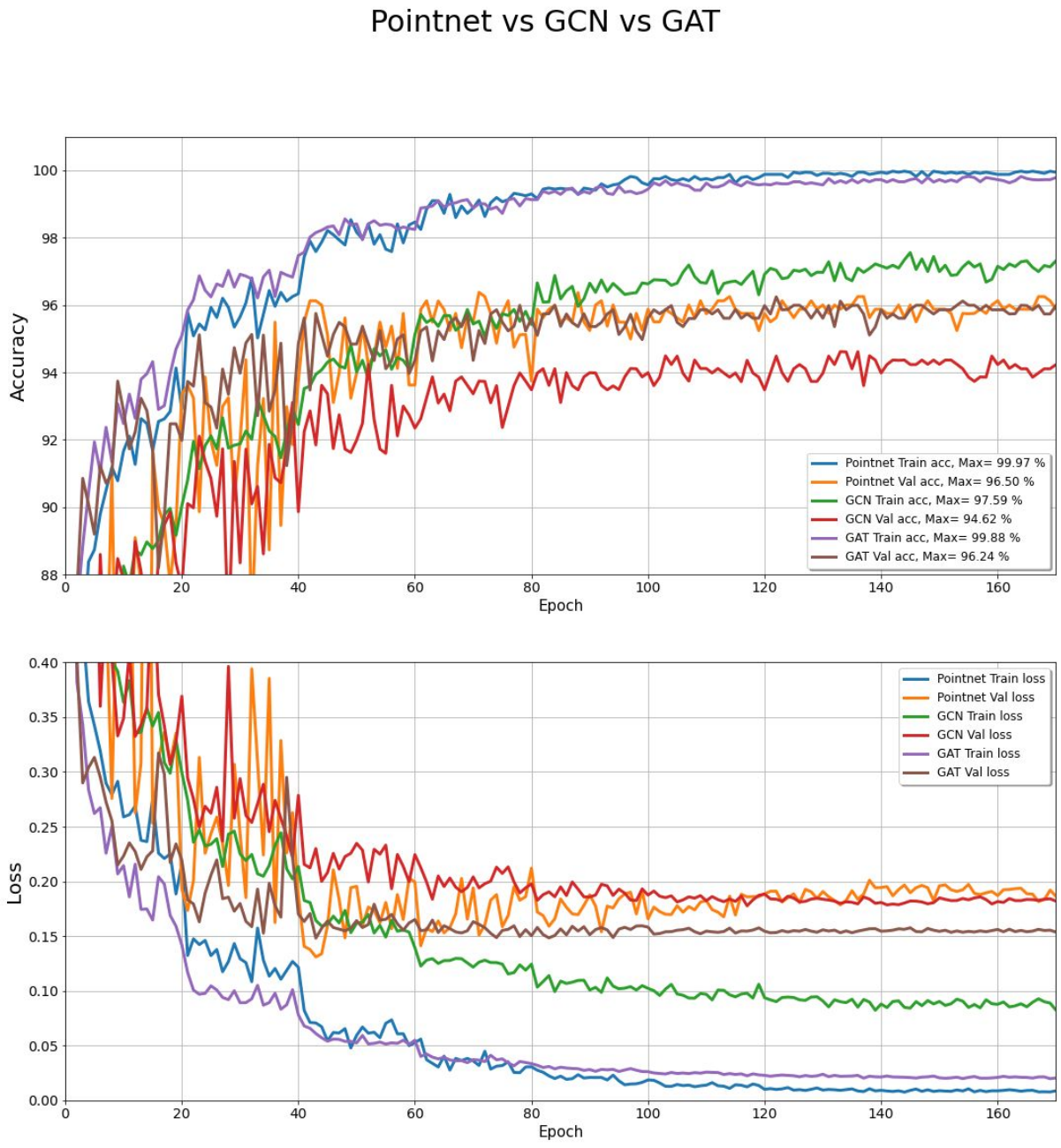




Conclusions

In this section, we will compare the best results obtained with our GNN architectures vs. our PointNet results and use the ModelNet “Test” dataset to obtain the test accuracy and the precision and recall statistics per class.

The following charts show the evolution of the accuracies and losses during the training phase of the PointNet, GCN, and GAT architectures. Out of all the GNNs tested, we have selected here the two architectures that performed the best in the test campaign described above: GCN with double capacity and GAT with 8 heads concatenated.



Until now, we have not used the Test dataset defined by ModelNet. It consists of 908 meshes that none of our networks has seen before. We sample the meshes and scale them to create the point clouds for Pointnet, and, after an additional step to create the edges, we create the graphs for the GNNs.

The TEST results are the following:

	Accuracy	Difference vs. Best	# Trainable Weights
PointNet	90.97%	-	3.463.763
Best GCN	89.21%	-1.76%	12.298
Best GAT	90.09%	-0.88%	173.834

In terms of accuracy the best network remains PointNet but our Best GAT architecture performs similarly to it.

Note that the number of trainable parameters of PointNet is almost 20 times higher than the GAT architecture with 8 heads. In terms of speed, our GNN networks require an intermediate step consisting of creating the graph. This is not required in PointNet.

Between GNNs, GAT performs slightly better than GCN at the expense of 14 times more parameters. Note that the training time of GATs is relatively long due to the extra computation needed to calculate the attention heads. GCN-based networks offer in this regard a quite interesting performance.

In terms of Precision and Recall of the Test dataset for each class and each network the results are the following:

		PointNet		Best GCN		Best GAT	
	Samples	Precision	Recall	Precision	Recall	Precision	Recall
Bathtub	50	100.0%	94.0%	100.0%	82.0%	100.0%	84.0%
Bed	100	96.1%	98.0%	92.4%	97.0%	94.2%	97.0%
Chair	100	99.0%	100.0%	98.0%	100.0%	97.1%	100.0%
Desk	86	71.6%	84.9%	68.3%	82.6%	74.5%	84.9%
Dresser	86	83.3%	87.2%	74.5%	84.9%	77.4%	83.7%
Monitor	100	100.0%	99.0%	99.0%	97.0%	98.0%	98.0%
Night_Stand	86	84.2%	74.4%	86.7%	75.6%	81.0%	74.4%
Sofa	100	95.2%	98.0%	97.0%	96.0%	97.0%	97.0%
Table	100	83.2%	74.0%	83.7%	72.0%	84.4%	76.0%
Toilet	100	99.0%	98.0%	98.0%	98.0%	99.0%	99.0%
	908						
Weighted Avg.		91.2%	91.0%	89.8%	89.2%	90.3%	90.1%

The patterns of the three networks in terms of Precision and Recall are similar. All three perform worse in identifying the classes: Desk, Dresser, Table and Night_Stand. The detailed results in terms of True and False Positives and False Negatives is given below.

	PointNet			Best GCN			Best GAT		
	True Positives	False Positives	False Negatives	True Positives	False Positives	False Negatives	True Positives	False Positives	False Negatives
Desk	73	29	13	71	33	15	73	25	13
Dresser	75	15	11	73	25	13	72	21	14
Night_Stand	64	12	22	65	10	21	64	15	22
Table	74	15	26	72	14	28	76	14	24

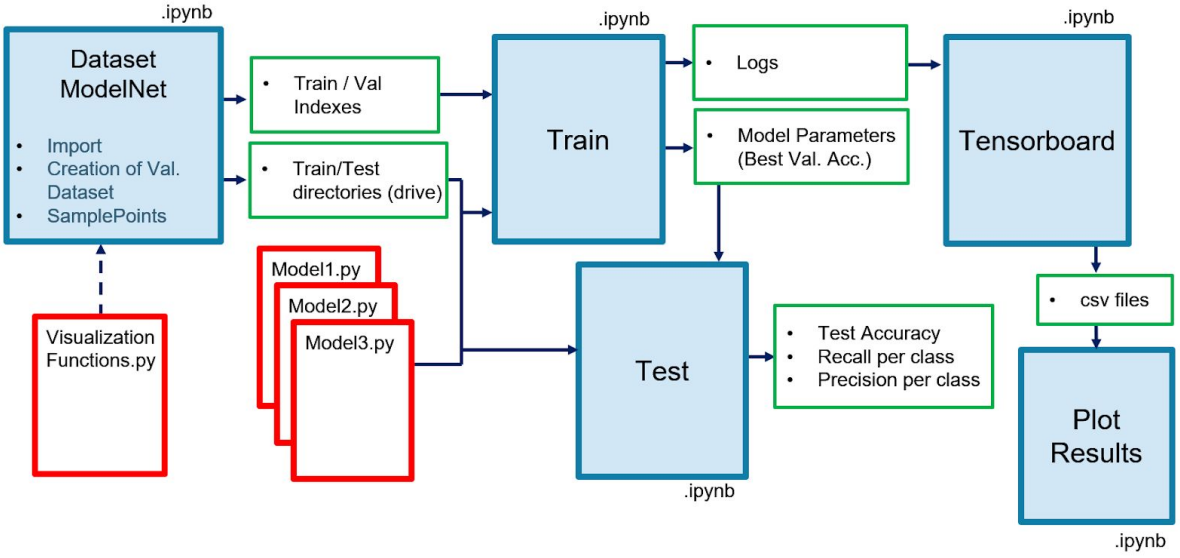
This could be due to our approach in scaling all the images before entering the model. After all, a Desk is a sort of Table, and a Dresser can be confused with a Night_Stand (or vice versa) if both images have the same size.

Future Work

After completing the development of two successful convolutional graph neural networks for object classification, we still consider that there is room for improvement. Several methodologies could yet be applied in order to achieve even higher classification performances. Below, a list of different strategies that could be followed is given:

- Try different k's (k-nearest neighbors)
- Using weighted loss for dataset categories imbalance
- Try the use of directed graphs.
- Using different and more extensive datasets such as ModelNet40

Code Structure



Repository Structure

The repository is structured as follows:

- architectures - folder containing all the architectures models.
- experiments - folder containing logs of all the experiments carried out.
- figures - folder containing all the plots (png format).
- Dataset.ipynb - colaboratory file containing the dataset retrieval and information.
- PointNet_train.ipynb- colaboratory file containing the train class.
- PointNet_test.ipynb - colaboratory file containing the test class.
- GNN_train.ipynb- colaboratory file containing the train class.
- GNN_test.ipynb - colaboratory file containing the test class.
- train_split.txt - txt file containing the indexes of the training data.
- val_split.txt - txt file containing the indexes of the validation data
- Visualization_functions.py - python file containing all the visualization functions.
- Tensorboard.ipynb - colaboratory file containing the plots.

Steps to run colabs

1. Create in Google drive a folder called **Proyecto.**
2. Create a folder called **Colabs** under Proyecto and copy there the following files: Dataset.ipynb, PointNet_train.ipynb, PointNet_test.ipynb,GNN_train.ipynb, GNN_test.ipynb, Tensorboard.ipynb and Visualization_Functions.py
3. Create the folder Architectures under Proyecto/Colab and copy all the architectures .py.

4. Run the Dataset Colab to download the ModelNet dataset and create the train, validation and test datasets.
5. Run the train colab, PointNet_train or GNN_train, to train the model selecting the architecture you want.
6. Once you train the model, open a test colab, PointNet_test or GNN_test, to test the model by selecting the architecture you want. Then, you also need to upload the best_params.pt file that contains the parameters of the model with the best validation accuracy.
7. Finally, use the tensorboard colab for visualizing train, validation and test curves.

References

[QIC16] Qi, C. Su, H. Mo, K. and Guibas, L. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation *arxiv:1612.00593.(2016)*

[ZHO18] Zhou, J. Cui, G. Zhang, Z. Yang, C. Liu, Z. and Sun, M. Graph Neural Networks: A Review of Methods and Applications. *CoRR* (*2018*)

[GAR16] Garcia-Garcia, A. Gomez-Donoso, F. Garcia-Rodriguez, J. Orts-Escolano, S. Cazorla, M. and Azorin-Lopez, J. "PointNet: A 3D Convolutional Neural Network for real-time object class recognition," *International Joint Conference on Neural Networks (IJCNN)*, Vancouver, BC, Canada, (2016)

[KIP16] Kipf, T.N. and Welling, M. Semi-Supervised Classification with Graph Convolutional Networks. arXiv:1609.02907

[VEL18] Veličković, P., Casanova, A., Lio, P., Cucurull, G., Romero, A., & Bengio, Y. (2018). Graph attention networks. 6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings https://doi.org/10.17863/CAM.48429