Group Invariant ML: Research Project at Imperial College, London

Meg Dearden-Hellawell and Hugo Robijns

Summer, 2024

Abstract

In a recent paper by a team at Cambridge University [Aggarwal et al., 2023], simple feed-forward neural networks were used to learn certain topological quantities of Calabi-Yau 7-manifolds. Through considering both basic ML techniques, as they did, but also group invariant methods, this short project aimed to reproduce and potentially improve their results. Ultimately, the change in accuracy was not statistically significant, as perhaps expected.

This document is aimed at providing a summary of the work achieved and topics learnt during the project. This was the first and shorter of the two projects completed over the summer weeks at Imperial, and served mainly as an educational exercise and introduction into the field of geometrical machine learning. As a result, it is written more as a collection of notes and observations for future reference, rather than a formal research report - it may therefore be light on context and references etc.

A big thank you to Daniel Platt from Imperial and Daattavya Agarwal from Cambridge for their time and help this summer!

1 Introduction

Neural networks (NNs) are computational models designed to mimic the human brain, and are used to find correlations in complex, large and seemingly unrelated datasets. Machine learning (ML) and NNs have found widespread applications over the past years, including facial recognition, weather forecasting etc.

The intersection between ML and geometry is a rich field, and was the focus of the summer at Imperial. This first project in particular concerned a recently published paper [Aggarwal et al., 2023], which used NNs to learn various features (Hodge numbers, Gröbner basis lengths etc.) of Calabi-Yau manifolds, to (for most of the features) great success. This is important since formally calculating these quantities can be computationally expensive, and often an accurate estimate can be valuable - the magnitude of certain Hodge numbers for example are relevant when it comes to evaluating a manifold's eligibility for string theory.

We in particular focused on learning a particular Sasakian Hodge number, $h^{2,1}$, and later the CN invariant - in this way, the input was the weight, a vector of length 5, and the output a real number. Interestingly, the problem is known to be invariant to the permutation of the input vector, but since the NN is not aware of this constraint, the original simple model that we (and the paper) trained does not necessarily have this quality - enforcing this is therefore a potential method to steer the NN down the right track and increase accuracy.

We experimented with 5 different permutation invariant architectures, based on the paper Deep Sets [Zaheer et al., 2018], the general definition for a permutation invariant function, and fundamental domain projections, as in [Aslan et al., 2022]. None of these methods greatly improved accuracy, however the project was still a success and we learnt important techniques applicable elsewhere, including our subsequent project this summer.

1.1 Invariance versus equivariance

A quick note on invariance versus equivariance - a lot of these methods and papers mention/highlight differences between these two definitions, i.e. if f is **invariant** to g:

$$f\left(g\left(a\right)\right) = f\left(a\right)$$

versus if f is **equivariant** to g:

$$f\left(g\left(a\right)\right) = g\left(f\left(a\right)\right)$$

Since in this problem our output is a single real number, and we are considering invariance/equivariance w.r.t permutation, these terms are equivalent in this context.

1.2 Code

The full repository, including links to the original dataset and .py files for all the architectures mentioned can be found in a GitHub repository **here**.

2 Methods

2.1 Deep Sets

The first two methods were based off a paper called Deep Sets [Zaheer et al., 2018].

2.1.1 Theory

In the paper it is shown that, if we represent a standard NN layer as $\boldsymbol{f_\Theta} = \boldsymbol{\sigma}(\boldsymbol{\Theta x})$ where $\boldsymbol{\Theta} \in \mathbb{R}^{M\times M}$ is the weight vector and $\boldsymbol{\sigma} : \mathbb{R} \to \mathbb{R}$ is a non-linearity such as the sigmoid function, then $\boldsymbol{f_\Theta} : \mathbb{R}^M \to \mathbb{R}^M$ is *permutation equivariant iff:*

$$\boldsymbol{\Theta} = \lambda \boldsymbol{I} + \gamma \left(\boldsymbol{1 1^T}\right) \tag{1}$$

where λ and γ are real parameters, $\boldsymbol{I} \in \mathbb{R}^{M\times M}$ is the identity matrix, and $\boldsymbol{1} = [1, \ldots, 1]^T \in \mathbb{R}^M$. The full proof can be found in the paper, but essentially this means that for any input vector we can construct these equivariant layers, which can also be stacked (see implementation section 2.1.2 for more detail).

It is also shown that a function $f(X)$ is invariant to the permutation in a set X iff it can be decomposed in the form:

$$f(X) = \rho\left(\sum_{x\in X} \phi(x)\right) \tag{2}$$

for suitable transformations ϕ and ρ. This is very similar to the expression discussed in section 2.2.1, and this is reflected in the similar architecture between these two models.

2.1.2 Implementation

method a: equation (1), equivariant model

- *equivariant layers*: each equivariant layer was designed according to equation (1), i.e. by summing two parts;
    - a simple convolutional part which introduces one parameter ($\lambda \boldsymbol{I}$):
    - and a second part where the input vector is pooled (in this case take the average of the elements of the input), the dimensions expanded (i.e. this real number from the previous pooling step is repeated/tiled to form a new tensor of the right output shape), then a parameter introduced ($\gamma \left(\boldsymbol{1 1^T}\right)$.
- *stacking layers and further training*: multiple such equivariant layers were stacked to build a network, with dense, hidden layers added to the end for further training.

method b: equation (2), invariant model

- *input transformation*: the input vector of length five, $(x_1, x_2, x_3, x_4, x_5)$ was mapped to a layer of length 5 with **no learnable parameters** - i.e. the input vector was essentially split into its 5 elements.
- *shared NN*: the same NN (ϕ) was run in parallel on the 5 elements.
- *pooling and further training*: the 5 outputs of the shared NN were summed together, and this output was then further trained (ρ).

2.2 General Invariance

2.2.1 Theory

A more general formula for permutation invariant function $\psi(x)$, where x is a vector of length n, is given by:

$$\psi(x) = \frac{1}{N} \sum_{g \in S_n} \phi\left(g \cdot x\right) \tag{3}$$

i.e. if you can decompose your function ψ as the sum/pooling of the output of a different function ϕ acting on each permutation of the input vector, then your function is clearly invariant to the permutation of the input vector.

2.2.2 Implementation

There were two methods in which this was implemented - either training ϕ, then averaging (method d), which was very simple to implement but naturally got low accuracies, or by training ψ directly (method c), which got higher accuracies but was slightly more complex. *Note that these methods were feasible since our group, S_5, only had 120 elements, and so this process was not too computationally intensive.*

method c: equation (3), training ψ

- *input transformation*: the input vector of length five, $(x_1, x_2, x_3, x_4, x_5)$ was mapped to a layer of size 120×5, representing all 5! possible permutations of the input vector. This transformation was **parameter-free**.
- *parallel NNs*: then, a shared neural network ϕ was applied to each of the 120 permutations in parallel. Each network processed one permutation of the input vector and produced a single real-number output.
- *aggregation and further training*: finally, the outputs from the 120 parallel networks were summed to produce the final output of ψ, which was trained further.

method d: equation (3), training ϕ

- *train a simple NN*: trained a simple feed-forward NN, ϕ, as in the paper, on the ordered input dataset.
- *average*: the output of the **overall model** (ψ) for a given input was then taken to be the average of the outputs of ϕ for all 5! permutations of that input. In this way, although ϕ is not invariant to the permutation of the input vector, ψ naturally is.

2.3 Fundamental Domain Projections

One way to force a NN to be group invariant is to map it to its fundamental domain [Aslan et al., 2022]. The paper provides of ways of deducing the fundamental domain for a given input, but for our input it was simply to sort the inputs in ascending order.

By sorting the input weights into ascending order, then training the NN on this, and then pre-processing any input data for prediction in the same way, we had a way of predicting Hodge numbers which is invariant under the permutation of the input weight.

Due to the fact that the data was already in the fundamental domain, this technique was trivial and redundant for this particular problem - it is essentially already part of our (and the paper's) original NN. *This is also a reason why our supervisors and us predicted from the start that these group invariant ML methods would likely not significantly increase the accuracy found in the paper.*

3 Results

Each model was trained and tested ten times, with the mean and standard deviation of the accuracy for these runs displayed in the tables below.

3.1 Sasakian Hodge numbers

Method	**Accuracy(%)**		**# learnable parameters**
	ordered inputs	**unordered inputs**	
vanilla	94 ± 4	38 ± 9	c.1.2×10^3
a	92 ± 7	92 ± 7	c.2.3×10^4
b	95 ± 3	95 ± 3	c.2.8×10^4
c	95 ± 4	95 ± 4	c.1.2×10^3
d	39 ± 9	39 ± 9	c.1.2×10^3

Table 1: Results for learning Sasakian Hodge numbers for each model. Here, *vanilla* specifies a simple feed-forward **regression** NN as used in the paper. Note that accuracy is also as defined as a paper (see section 4.1).

3.2 CN invariants

Method	**Accuracy(%)**		**# learnable parameters**
	ordered inputs	**unordered inputs**	
vanilla	6.0 ± 0.2	6.0 ± 0.2	c.1.6×10^3
a	6.2 ± 0.6	6.2 ± 0.6	c.2.3×10^4
b	6.2 ± 0.4	6.2 ± 0.4	c.2.2×10^4
c	6.4 ± 0.5	6.4 ± 0.5	c.1.6×10^3
d	1.2 ± 0.7	1.2 ± 0.7	c.1.6×10^3

Table 2: Results for learning the CN invariant for each model. Here, *vanilla* specifies a simple feed-forward **classification** NN as used in the paper. Note that since this is now a classification model, accuracy is defined differently to above (see section 4.1).

4 Discussion

4.1 Methods of Determining Accuracy

For fair comparison, the definition of accuracy is taken from the paper, i.e. for learning the Sasakian Hodge number, since a regression NN was used, the accuracy is defined as follows:

- a bound is defined - in this case, it is 5% of the range of the values of the training set Hodge numbers.
- the prediction is said to be correct if the difference between the actual Hodge number and the prediction is less than this bound, else it is incorrect.

For CNI, the model was turned into a classifier, and the accuracy is more simple - the percentage of correctly classified CNI.

4.2 Tuning

Hyper-parameter tuning was carried out qualitatively by changing various parts of the architecture (size and number of layers etc.) and noting changes in accuracy. In future, it would be beneficial to formalise this process, perhaps through using software that can sweep through the NN.

4.3 Discussion of Results

Simple t-tests were carried out to evaluate if there was any statistical difference between the methods, especially compared to the method of the paper (which we have called *vanilla*). No statistical difference was found compared to the paper for methods b and c, with d showing low accuracies as expected.

References

[Aggarwal et al., 2023] Aggarwal et al. (2023). Machine learning sasakian and g2 topology on contact calabi-yau 7-manifolds. *Physics Letters B*, 850.

[Aslan et al., 2022] Aslan, B., Platt, D., and Sheard, D. (2022). Group invariant machine learning by fundamental domain projections.

[Zaheer et al., 2018] Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R., and Smola, A. (2018). Deep sets. Accessed: 2024-07-09.