CSE663: Meta-Learning
Final Project Report
Neural Complexity Measures

Rishabh Gupta
rishabh19089@iiitd.ac.in

Himanshu Singh
himanshus@iiitd.ac.in

April 28, 2022

1 Paper Overview

The paper [3] introduces a novel framework for predicting the generalization capabilities of a task learner. This is done by meta-learning a model (called the Neural Complexity Model) that takes the task learner's predictions as input and gives a scalar output which estimates the value of the generalization gap. This is nothing but an estimate of the difference between the testing loss and training loss of the task learner. This value acts as a regularization term and can be added to the loss of the task learner to improve its generalization capabilities.

The NC model makes use of the training data (X_{tr}, X_{te}, Y_{tr}) as well as the predictions of the task learner (the task learner is denoted by h) on the train and test set as the input and uses the interaction between them to compute an estimate for the generalization gap. The interaction is modelled by an attention layer, where the query is created by the encoded representations of the training data, the key by the representations of the training data and the value by concatenating the representation of the training data with the training labels. This is used to compute the attention outputs using the standard self-attention formulation:

$$output = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

where Q, K, V represent the query, key and value matrices respectively and d_k is the dimension of the embedding. This basically captures the context by computing the interaction between each test sample (as captured by Q) and each training sample (as captured by V) which is then used to interact with the training labels (as present in V). The output of the attention layer is then pass through a decoder to get the final value of the gap estimate.

The architecture of the NC model differs slightly for the classification task as a bilinear layer is added to compute the value matrix V. The bilinear layer takes the encoded representations of the training data, along with the concatenation of the training labels, a vector of 1s and the train loss. The vector of 1s acts as a residual connection for the training representations through the bilinear layer. To save compute, the authors make use of a memory bank, where they store the snapshots of the necessary information for training the Neural Complexity model and reuse it as required.

We also noticed a couple of errors in the paper: As stated repeatedly in the paper, the output of the NC model will depend upon the task learner h, as the generalization capabilities of any model must depend on the model in some manner. In Equation 8 (as shown in Figure 1) on the LHS, the input to the NC model contains $h(X_{tr})$ and $h(X_{te})$, which contains some information pertaining to the task learner h. However, *this information is not utilized by the NC model at all* as per the equations 6, 7 and 8, which is a mistake made by the authors. Another small mistake made based on the notation occurs in Section 3.3, subsection Classification (just above equation 9) where the authors refer to the dimensions of Y_{tr} as $m' * c$ instead of $m * c$ as earlier m had been defined to be the number of training samples, and m' the number of testing samples.

$$\mathrm{NC}(X_{\mathrm{tr}}, X_{\mathrm{te}}, Y_{\mathrm{tr}}, h(X_{\mathrm{tr}}), h(X_{\mathrm{te}})) = \frac{1}{m'} \sum_{i=1}^{m'} f_{\mathrm{dec}}(e_{\mathrm{att}})_i \in \mathbb{R}.$$

Figure 1: Equation 8

2 Our Implementation Details

Due to the multi-task multi-class nature of the dataset, we had to make some changes in the implementation of the models along with other changes. First, to process the data, we normalize it to a standard distribution when loading it. We also make sure to sample the tasks in order of the day as defined by the file name i.e. making sure that a task with a lower day comes before a task with a higher day. We also use the first 20 days (100 pickle files) as the meta-training set and the remaining 6 days as the meta-testing set.

For the task learner, we use a stack of MLPs as the basic architecture for the flat dataset, the outputs of which are passed into 5 different MLPs, one of which is for regression (having 6 output neurons) and 4 of which are for classification (each having 4 output neurons). For the time series data, we use an LSTM/GRU instead of the stacked MLPs in the architecture to better model sequential data. Regression is trained with the MSE loss and classification is trained with the cross entropy loss.

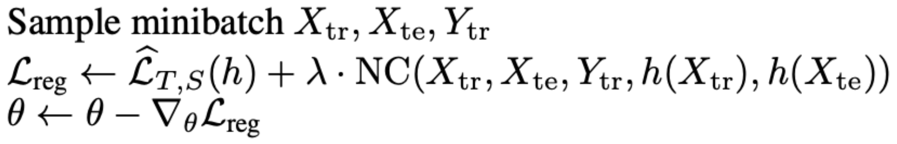
As described in the paper, we use a stack of MLPs for the NC encoder. This contains the linear layer, followed by the ReLU activation, which is followed by a batch normalization layer. We also add a layer of dropout to make sure that the model does not overfit to the training data. Since the paper suggested using a bilinear layer (which turned out to be the most critical component from the ablation studies) for classification, and since our setting is that of multitask classification and regression, we adopt the bilinear layer in our architecture to compute the value matrix V. As pointed out above, the paper did not take into account the predictions of the task learner ($h(X_{tr})$ and $h(X_{te})$). To rectify that, we always concatenate the predictions of the task learner to the input matrix. That is, we concatenate the train predictions ($h(X_{tr})$) to the the training set (X_{tr}) and the test predictions ($h(X_{te})$) to the the test set (X_{tr}) wherever applicable while computing the Q, K and V matrices. This allows us to incorporate the task leaner itself in some manner when training the NC model. In the bilinear layer, along with X_{tr}, we use a concatenation of Y_{tr}, a vector of 1s, L_{tr} (the training loss) and $h(X_{tr})$ (the train predictions).

We initialize the task learner again for every epoch and train it for all tasks. We calculate the losses for all the tasks and add them together to form the total loss. This loss is back-propagated to train the task learner and their mean is used to calculate the gap to train the NC model. We also clip the gradients of the NC model to make sure the training is smooth.

3 Further Ideas

3.1 Uncertainty Aware Gap Estimation

When training the task learner, the authors set the weight of the NC estimate, *lambda*, initially as 0 and then after a certain number of steps to 1 (as shown in Figure 2).



Sample minibatch $X_{\text{tr}}, X_{\text{te}}, Y_{\text{tr}}$

$\mathcal{L}_{\text{reg}} \leftarrow \widehat{\mathcal{L}}_{T,S}(h) + \lambda \cdot \text{NC}(X_{\text{tr}}, X_{\text{te}}, Y_{\text{tr}}, h(X_{\text{tr}}), h(X_{\text{te}}))$

$\theta \leftarrow \theta - \nabla_{\theta} \mathcal{L}_{\text{reg}}$

Figure 2: Snapshot from Algorithm 1 in the paper

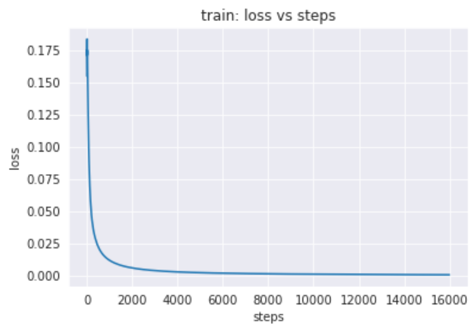
This leads to the question of deciding when exactly (which step) to make the value of the parameter λ as 1. We came up with two ways of answering this question (and implemented 1 of them). The first way is to simply parameterize alpha to be estimated by a neural network. That is, given the inputs to the NC model and the output prediction of the NC model, we can train a neural network which to output the value of λ. However, this suffers from the disadvantage of the capability of the chosen model in successfully modelling the optimal values for λ.

The second (and perhaps more interesting) way that we came up with to do this is to use the uncertainty of the model estimates to find the value of the parameter λ. Ideally, Bayesian inference models should be used to model this uncertainty, but they are compute expensive and hard to train. Thus, we use Monte Carlo Dropout [1] to run multiple simulations for each sample while training and take the standard deviation of the model estimates as the uncertainty value υ. Since we want to update λ to be closer to 1 as the training goes on, we bias it towards 1 as the training steps increase by exponentiating the uncertainty value with the number of steps that have occurred t to make the uncertainty estimate smaller and thus, λ closer to 1. More formally,

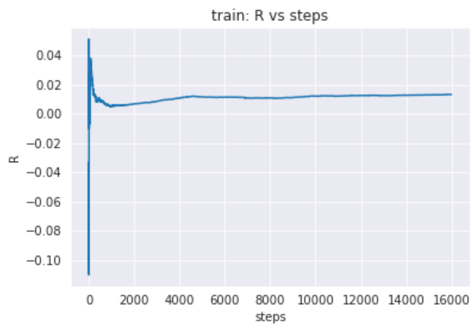
$$\upsilon = \sigma(NC(X_{tr}, X_{te}, Y_{tr}, h(X_{tr}), h(X_{te})_i) \; \forall i \in [1, n\_simulations]$$
$$\lambda = max(0, min(1, (1 - \alpha * \upsilon^{\lfloor t/\kappa \rfloor + 1})))$$

where α and κ are two hyper-parameters we use to control the degree of influence of the uncertainty and the current training step.

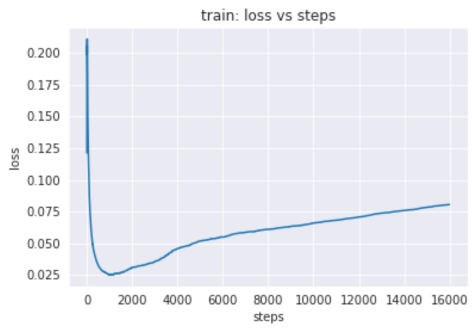
Note that there are other ways to model uncertainty using Monte Carlo sampling like using spatial dropout, DropBlock [2] (when using CNNs) amongst others, however due to a lack of time we were not able to try them. For training, we set the value of α as 0.8 and the value of κ as 1000. We did not tune these hyperparameters due to the large run time of the model and our time constraints, however, these can be tuned to improve the performance significantly. Due to the larger training time, we were not able to run the uncertainty aware model on the time-series data.



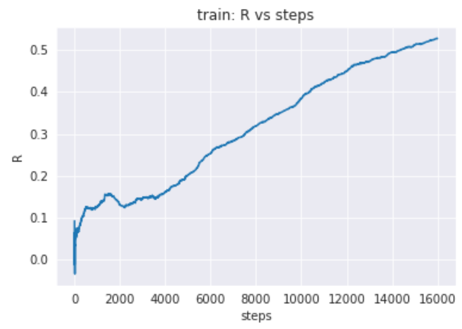
(a) Train Loss of the NC Base model



(b) Train R of the NC Base model



(c) Train Loss of the Uncertainty Aware NC model



(d) Train R of the Uncertainty Aware NC model

Figure 3: Training Loss and R of the NC Base and Uncertainty Aware models on the Flat dataset

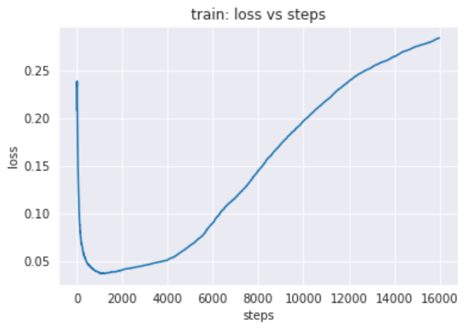
3.2 Combining the predictions of the task learner with input data

Another possible change that can be made is to add more interaction between the prediction of the task learner and the input data when combining them to give as input to the NC model. A simple way of doing this would be to use something like a bilinear layer to compute their interaction instead of just concatenating them and passing them into the NC model. However, due to time constraints, we did not get the opportunity to practically run this idea.

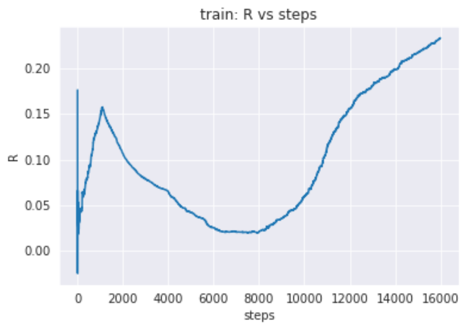
4 Experiments

We perform multiple experiments on both the flat (cs) dataset and the time-series (ds) dataset. We train each model for 10 epochs on an NVIDIA A5000 GPU. On the flattened data, we also do an ablation study of the impact of various layers present in training the neural complexity model. On the flattened dataset, we perform the following experiments:

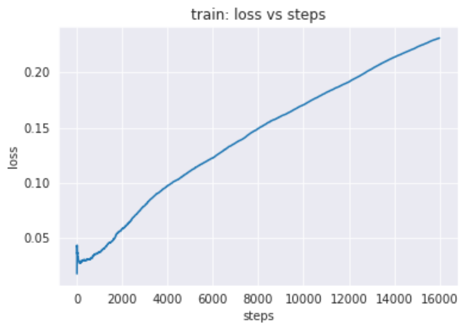
- Not Using NC
- Using the base version of NC(NC Base)



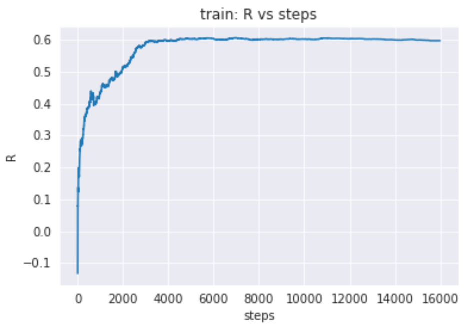
(a) Train Loss of the NC LSTM model



(b) Train R of the NC LSTM model



(c) Train Loss of the NC GRU model



(d) Train R of the NC GRU model

Figure 4: Training Loss and R of the NC LSTM and GRU models on the Time-Series dataset

Dataset	**Method**	Regression					
		MSE					
flat	without nc	**0.0532**	**0.0396**	**0.0351**	**0.0759**	**0.0512**	**0.0527**
	nc base	0.2853	0.3383	0.4024	0.2455	0.2223	0.4885
	nc bigger	0.6624	1.2099	0.6232	1.4001	0.6648	0.896
	nc without bilinear	0.7797	0.9517	0.8449	1.3765	0.9429	1.0798
	nc without h_pred	0.653	0.5273	0.3047	0.5143	0.5073	0.4384
	nc without loss	0.2953	0.2611	0.2461	0.2742	0.2559	0.2574
	nc with bilinear addition	1.0508	1.6263	0.853	0.8873	1.3194	0.671
	uncertainty aware nc	0.6776	0.8195	1.131	0.8889	1.346	1.0543
timeseries	without nc	**0.0255**	**0.0276**	**0.0305**	**0.03**	**0.0248**	0.0312
	nc lstm	0.0334	0.0387	0.0414	0.039	0.034	**0.0308**
	nc gru	0.4948	0.6299	0.2993	0.3994	0.3908	0.4518

Table 1: Results for Regression for all methods across both datasets

Dataset	Method	Classification							
		D1		D2		D3		D4	
		Acc	**F1**	**Acc**	**F1**	**Acc**	**F1**	**Acc**	**F1**
flat	without nc	**0.41**	**0.44**	0.52	0.56	0.24	0.24	0.42	**0.43**
	nc base	0.38	0.41	0.44	0.48	0.23	0.21	0.36	0.38
	nc bigger	0.19	0.2	0.12	0.09	0.27	0.29	0.23	0.26
	nc without bilinear	0.32	0.35	0.39	0.46	0.23	0.22	0.37	0.38
	nc without h_pred	0.29	0.33	0.37	0.42	0.27	0.28	0.38	0.36
	nc without loss	0.25	0.3	0.17	0.23	0.22	0.24	0.29	0.32
	nc with bilinear addition	0.35	0.38	0.3	0.37	0.21	0.17	**0.45**	0.39
	uncertainty aware nc	**0.41**	0.43	**0.55**	**0.58**	**0.36**	**0.32**	0.34	0.34
timeseries	without nc	0.37	0.4	**0.57**	**0.57**	0.28	0.27	0.31	0.3
	nc lstm	**0.42**	**0.44**	0.56	**0.57**	0.33	0.32	0.27	0.28
	nc gru	0.37	0.4	0.51	0.53	**0.36**	**0.33**	**0.35**	**0.37**

Table 2: Results for Classification for all methods across both datasets

- Using a bigger version of NC(NC Bigger)
- NC without using the bilinear layer
- NC without using $h(X_{tr}$ in the bilinear layer
- NC without using L_{tr} in the bilinear layer
- NC where we project and add the output of the bilinear layer to the query matrix
- Uncertainty Aware NC (using NC Base)

Due to a lack of time and resources, we do not perform all the ablations on the timeseries dataset. We perform the following experiments:

- Not using NC
- Using NC with an LSTM (in the task learner)
- Using NC with a GRU (in the task learner)

To measure the impact of the Neural Complexity model on the task learner, we report MSE for the 6 regression tasks and accuracy and weighted F1 for the classification tasks. We use weighted F1 as it is more representative of the actual performance due to the imbalance in the classification data.

5 Results and Analysis

The plots for training the NC model are present in Figure 3 and Figure 4. In Figure 3, we have plotted the train loss and R-score for the base NC model (without uncertainty) and the uncertainty aware NC base model. We can see that the training loss for the uncertainty aware model starts to increase after a few steps, which would be countered by tuning the parameters more carefully (especially setting a lower learning rate and proper initialization). The R-score for the uncertainty aware model is increasing when compared to the base model.

The results for regression are given in Table 1 while the results for classification are given in Table 2. From the regression results, we can see that the model trained with NC (in any form) does not perform well on the flat dataset as the task learner trained without neural complexity easily beats it. On the time-series dataset, although the model trained without NC is generally better (for 5 out of the 6 tasks), the NC-LSTM model is very close by and performs better on 1 task. We hypothesize that the poor performance of using the NC model while training the task learner is due to its inability to effectively compute the gap in a multitask setting, especially since there are 10 different tasks which are of two types (regression and classification). Perhaps a different way to train the NC model would have been to compute the gap for each task separately and adding it to the loss for each task (instead of taking the mean of each task and adding the loss). We also note that the uncertainty aware method is not useful in the regression tasks as it is worsening the performance (perhaps training in a better manner would have helped but this is not experimentally known).

For the classification task, we note that the uncertainty aware NC model performs on par and in some cases better (except D4) than the method without NC on the flat dataset. We must note that we set the value of α to 1 for testing (using the model trained with $\alpha = 0.8$) as the trained model should be more confident about its estimates. Perhaps training the uncertainty model with $\alpha = 1.0$ might have resulted in even better performance. This demonstrates the usefulness of uncertainty estimation, especially for classification tasks. Using different methods to model the uncertainty rather than just MC Dropout can also yield interesting results. One disadvantage of uncertainty estimation is the marked increase in training time (around 4-5x training time of the NC base model). On the time-series data, the LSTM and GRU models both perform at par with or better than the method without NC, thus showing the usefulness of NC regularization, especially on this dataset.

From the ablations, we can see that all of the components involved in the NC base model are critical as the performance decreases when we remove any of the components. The biggest decrease in performance for regression comes when we remove the bilinear layer, while for classification the removal of the train loss in the bilinear input seems to affect it most significantly. Another interesting observation is that increasing the number of parameters in the task learner and NC model leads to a decline in the performance of the NC model. We observe this in the NC bigger model (where the task learner and the NC model itself had more parameters) which performs poorly on both the regression and classification tasks. We also checked this for the time-series dataset, where we trained the task learner (with GRU) with two different configurations. The configuration with more parameters gave poorer results, demonstrating the inability of the NC model to scale with the size of the task learner.

References

[1] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning, 2015.

[2] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V. Le. Dropblock: A regularization method for convolutional networks, 2018.

[3] Yoonho Lee, Juho Lee, Sung Ju Hwang, Eunho Yang, and Seungjin Choi. Neural complexity measures, 2020.