Deep Learning for Symbolic Mathematics Partial Replication

Gyung Hyun Je

Abstract

The report describes the process of partially replicating the results from Deep Learning for Symbolic Mathematics paper by G. Lample and F. Charton. Using a seq2seq model with an attention layer, the project trains the model to perform symbolic integrations given a non-numeric mathematical expression with operators and variables.

The main steps of the project implementation are two-fold: first, the project runs model experiments with five different sets of hyperparameters to choose the best model. Second, the project compares the model results quantitatively and qualitatively; specifically, if the mathematical expressions outputted by the model provide correct solutions. Key technical challenges include the resources required to fit the large dataset size used by the authors of the original paper and achieving a similar accuracy using a non-transformer based model. To compensate for the difference, the project provides a hyperparameter tuning class to facilitate a faster model experiment using different sets of hyperparameters.

The best model produces 80% accuracy on the validation set, which is lower than the 93%-95% accuracy achieved by the original paper; two main hypotheses are 1. the dataset size difference and 2. the model architecture. The report discusses in detail the reason for the gap.

1. Introduction

Many mathematical frameworks such as WolframAlpha provide a tool to conduct symbolic calculation of mathematical expressions. The original paper [1] by Lample and Charton shows that a simple Transformer based seq2seq model can perform symbolic function integration and differentiation with high accuracy. The goal of the project is to replicate the result of the integration. In the process, the replication of the original paper faced a few challenges. Three main challenges include:

1. *Limited resources to fit the entire dataset.* The original paper used 45M training samples for integration. Due to the limited RAM resources, the project uses the top 20M rows out of 45M training samples. The selected data tends to contain shorter expressions – this decision was made to simplify the model task given the limited data.
2. *Building a transformer model.* The initial goal was to build the transformer model as in the original paper. However, the main technical difficulty was correctly building a model that handles changing dimensions of the input sequences. Instead of implementing a transformer model, the replication focuses on a seq-2-seq model with an attention layer. To maximize the performance given the change in the architecture, the project uses a custom built model experiment class to run 5+ model experiments to optimize the model with the best hyperparameters and choose the best model.

2. Summary of the Original Paper

2.1 Methodology of the Original Paper

The approach of the original paper is two-fold.

First, the original paper generates random mathematical expressions that are differentiable or integrable and use the generated expressions as training data.The algorithm used to generate the expressions uniformly selects the operations (i.e. binary expressions such as addition and unary expressions such as sine), represent the generated expressions as prefix readings of a tree, and feed the sequences into the model. Authors of the original paper note that randomly and uniformly generating the expressions is key, because it allows the model to see a wide range of expressions that do not have a high bias towards a broad tree or a left or right deep tree structure. Such variety would ensure the model has good generalizability. One of the key ingenuity of the original paper comes from representing mathematical expressions in a tree form that maps one to one with the input. The standardized way of reading the expressions helps the model learn dependencies among variables and operators.

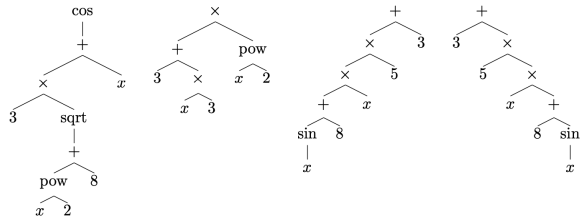


Figure 1: Examples of mathematical expressions represented in a tree form; the algorithm in the original paper generates these expressions with the same probability. This figure is from the original paper.

For the integration task, the original paper experiments and compares different data generation methods. One data generation method tends to output a short input and a long solution (FWD method and IBP method), the other one a long input and a short solution (BWD method). A combined method produces the best representation of the problem space. The project uses the dataset published by the authors of the paper [3].

Second, the original paper builds the model to train the data on. The trained model is a transformer seq2seq model with 8 attention heads, 6 layers, and a dimensionality of 512, batch size 256, with Adam optimizer with learning rate of 10-4.

2.2 Key Results of the Original Paper

The figure below is the original paper showing the performance of the model on the held-out test set of 5000 equations:

	Integration (FWD)	Integration (BWD)	Integration (IBP)	ODE (order 1)	ODE (order 2)
Beam size 1	93.6	98.4	96.8	77.6	43.0
Beam size 10	95.6	99.4	99.2	90.5	73.0
Beam size 50	96.2	99.7	99.5	94.0	81.2

Figure 2: model on integration and the differentiation tasks. The project uses the FWD dataset for the integration task (first column).

For integration, the model achieves high accuracy on the held-out set with beam size 1, but differentiation requires a larger beam size to achieve a higher accuracy. The authors of the original paper observe that increasing the beam size, which in turn increases the number of hypotheses the model generates, has a significant improvement in the model accuracy. Achieving the performance of the integration task using FWD dataset is the original goal of the project.

3. Methodology

At a high level, the project builds a seq2seq model with an attention layer and the integration dataset provided by the authors of the original paper to train the model to perform symbolic integration. Due to the limited resources, the project uses up to 20M out of 45M training samples. Unlike the original paper, the model used to fit the training data is a seq2seq model with attention instead of a transformer based model due to the aforementioned technical difficulty mentioned in the introduction. To boost the accuracy of the seq2seq model, the project runs experiments with different hyperparameters and tests using different encoder layers (GRU or LSTM).

3.1. Objectives and Technical Challenges

There are two main goals of the project.

One, successfully build a seq2seq model with attention. To do this, the project refers to Tensorflow's online tutorial on building the seq2seq model [2]. The project writes custom variations and explicitly quotes the source if using the code directly from the tutorial.

Second, build a custom model experiment class for seamless hyperparameter tuning. Instead of manually resetting the hyperparameter, the project aims to make it easier to produce multiple model versions trained on different hyperparameters and compare the result.

Besides the technical challenges mentioned in the introduction, the main technical challenge with the modified plan was integrating the custom functions to the existing Tensorflow methods. Tensorflow methods and function attributes would be rendered incompatible with a slight modification in the environment or installation of other packages that broke the dependencies.

Another technical challenge was securing the GPU resource to run multiple model experiments. The model training portion ended up using the Google Colab GPU rather than the VM GPU due to

the several failed attempts to connect to the instance due to resource limitations.

3.2.Problem Formulation and Design Description

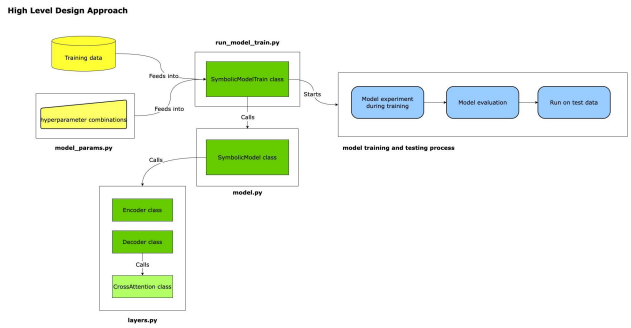


Figure 3: high level overview of the project implementation

The design workflow shows how the data, Python model objects, and the actual training steps fit together. Details of model architecture is in section 4.2, training steps in section 4.3. At a high level, there are two main steps.

First step is model experiments and training. First, it initializes the SymbolicModelTrain class to set the appropriate batch size and run model experiments with different sets of hyper parameters. Different combinations of hyperparameters are stored in model_params.py. Each i-th trial has the following dictionary format:

***trial_i**: {*
epoch: # training epoch
optimizer: model optimizer
num_units: model dimension
is_gru: True/False
num_heads: # attention heads
}

This format was chosen to make it easier for the user to change hyperparameters in one place (at model_params.py that the training class reads) without having to manually update it, as the manual update often introduces unforeseen errors when running multiple experiments. The train_model() function takes the parameters and constructs a new model at each trial. The model experiment class stores the fitted model and the training history in a dictionary as a class attribute, with the following key and value format:

***trial_i**: {*
trained_model: the model object
history: training history object
}

Second step is the evaluation of model accuracy and loss over epoch. Accessing the model and the training history from the class attribute allows easy access to the several models trained on different hyperparameters in the first step. Based on the evaluation, the best model can be picked to fit the test data on the model.

4. Implementation

The implementation section walks through the deep learning network architecture, the detailed end-to-end software design, and dataset used in the project, including the walk-through of the data cleaning process.

4.1 Deep Learning Network

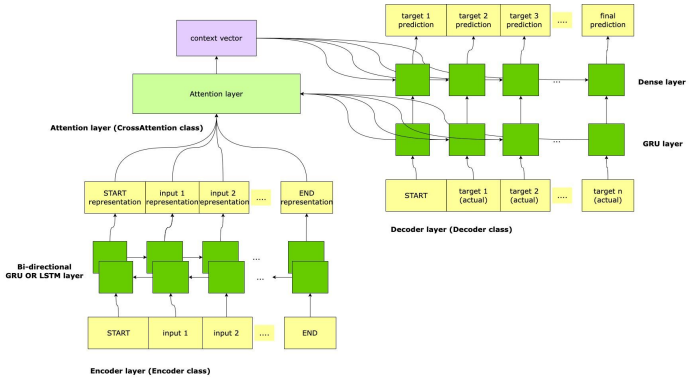


Figure 4: seq2seq model architecture used in the project

Model architecture

For the project, the seq2seq model with attention has been chosen since the task of translating the mathematical expressions into the corresponding expressions is analogous to the task of machine translation of language. The seq2seq with attention consists of encoder and decoder layers, and attention layer is used in the decoding phase to selectively pay attention to more relevant information from the encoder.

Encoder layer consists of a bidirectional RNN layer to provide access to both right and left contexts. The RNN layer either uses GRU or LSTM depending on which one produced a better result during model experimentation. Once the input sequence is tokenized and processed on the encoder RNN layer, attention score is calculated at the decoder layer. Number of attention heads is also a hyperparameter tested during the model experiment phase.

Unlike the encoder layer, the decoder layer is a unidirectional GRU layer because the model will not have access to the full sequence during the test period. During training, the decoder is given a correct target token at the next time step, regardless of its prediction at the previous time step.

Once the decoder and the encoder inputs are processed in the attention layer, the final dense layer using tanh activation function returns a prediction. An experiment with sigmoid activation function produced a lower overall accuracy on all trials, so the results were discarded and tanh function was used instead.

The model uses batch size of 128, early stopping with patience set to 5 to avoid overfitting, 100 steps per epoch for training steps and 20 steps per epoch for validation steps. A custom masked loss function using sparse categorical cross entropy is implemented for model training. A custom loss function significantly improves the model performance compared to the base sparse categorical cross entropy loss function. This is because the masked loss function ignores the padding in the output sequence. Due to the varying input and target sequence size, not using a masked loss function significantly hurts the model performance. The project also experiments the Adam and RMSProp optimizers during the model experiment phase.

Training and validation Data

The project uses the full validation data provided by the authors of the paper, but the training data is about 40% of the original training data due to the memory restriction and training speed. The model training uses 20M rows for training and 10,000 rows for validation.

The input and the out data for symbolic mathematical expressions are prefix reading of a tree, where nodes are the operators and leaves are the variables. Each operator has a unique corresponding symbol; for instance, “sub” maps to subtraction operator, “mul” to multiplication, “pow” to power and so on. The input expression "sub Y' x" translates to Y'-x = 0, equivalently to Y' = x. The corresponding output expression is "mul div INT+ 1 INT+ 2 pow x INT+ 2", which translates to "multipy [(divide 1 by 2) and (x to the power of 2)]", which is 1/2(x^2), a correct solution solving for Y. It is important to note that tokenization of the input and output sequences should preserve essential information for function integration. For example, the plus sign in the "INT+" should be kept to distinguish between a positive and negative integer. A single quote character that appears with Y signifies the derivative form of Y. The failure to preserve this information would produce an inaccurate result, since Y’ and Y are two very different tokens in the input sequence. The custom lower_and_strip_punctuation in the run_model_train.py script handles the correct sequence cleaning.

4.2 Software Design

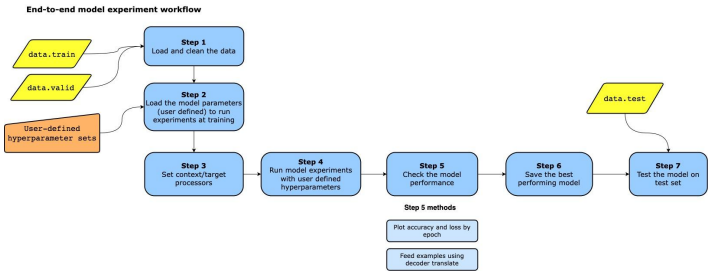


Figure 5: end-to-end model experimentation steps

The figure shows the end-to-end model experiment workflow from loading data to testing different sets of hyperparameters and choosing the best model to run the test data on.

Step 1 and 2 initializes the SymbolicModelTrain class and loads the necessary data into the class.

Step 3 sets the context and target processors. The processors correctly clean and tokenize the input and output sequences. They provide the mapping necessary for converting the sequences to tokens and tokens back to the sequences.

Step 4 is the main training phase that produces models run on different sets of hyperparameters. The main purpose of this step is to set up a pipeline for

model experiments, without having to manually reset the model hyperparameters.

The high level steps are as follow:

For each user defined model parameter set:

Step i: Load hyperparameters specified by user

Step ii: Initialize SymbolicModel

Step iii: Compile the model with given optimizer

Step iv: Run model fit with specified epoch

Step v: Store the trained model and training history

Details of step 4 and how the i-th trial builds the model with either GRU or LSTM layer is illustrated in the following figure:

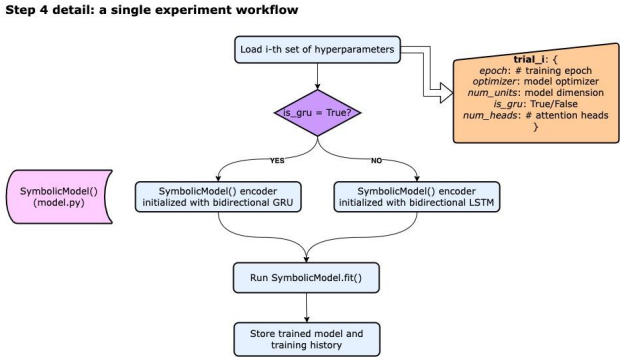


Figure 6: how the step 4 above builds a model with GRU or LSTM layer

Once the training is done with every set of hyperparameters, test the model by plotting the accuracy and the loss. The training phase (1 through 5) can be run in the run_all function in run_model_train.py script, but it is possible to call the functions individually to check the output at each step. The jupyter notebook that runs the model training and evaluation calls the methods individually.

5. Results

5.1 Project Results

The sets of hyperparameters that were tested and achieved the final loss and accuracy is listed below:

Trial	Epoch (early stopping if applicable)	Optimizer	Model dimension	GRU/ LSTM	Attention heads	Final loss	Final accuracy
Trial 1	100 (stopped at 91)	Adam	64	GRU	4	0.3397	0.7467
Trial 2	100	RMSprop	64	LSTM	4	0.3786	0.7516
Trial 3	100 (stopped at 85)	Adam	256	LSTM	8	0.3619	0.8041
Trial 4	100 (stopped at 54)	Adam	256	GRU	8	0.3535	0.7831
Trial 5	100 (stopped at 68)	Adam	512	GRU	8	0.3605	0.8045

Table 1: shows the model experiment results

The following graphs show loss and accuracy over epoch. To check comparison between the training and validation loss/accuracy, refer to the Appendix A:

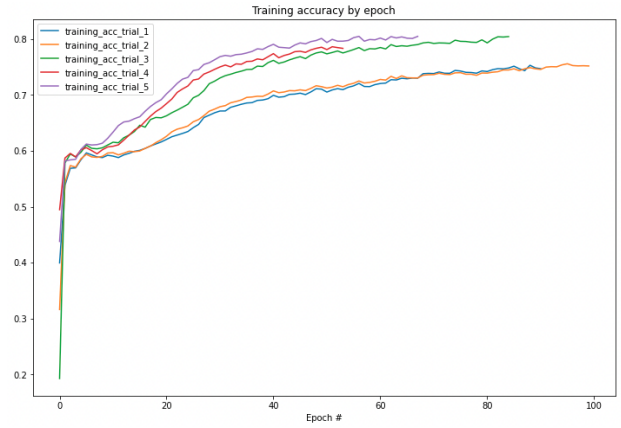


Figure 7: model training accuracy by epoch

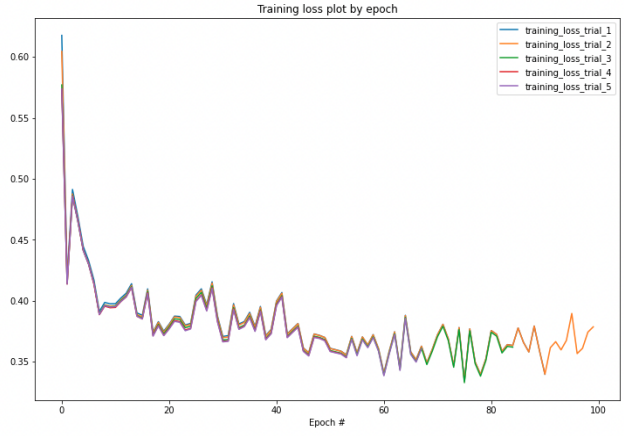


Figure 8: model loss by epoch

The models with smaller dimensions and fewer attention heads required more epochs to converge,

and they overall have lower accuracy than models with higher dimensions and with more attention heads. Between trial 3 and trial 4 in which the only difference was the use of GRU or LSTM, and trial 3 with LSTM had a better performance. Between trial 5 and previous trials, the model had the best accuracy but it was comparable to trial 3. Increasing the number of units improved the performance by 5+% when the units were increased from 64 to 256, but the difference between 256 and 512 was not very significant.

Qualitative assessment of the models was difficult, as some models did better on some tasks, but there was no apparent trend. For example, model 3 outputs a perfect prediction given the sequence "sub Y' x" (y' = x). The solution is “mul div int+ 1 int+ 2 pow x int+ 2” (equivalent to 1/2*(x^2)). On the other hand, the model 4 returns a long sequence that contains the correct solution (equivalent to 1/2*x^2) but other unnecessary terms, and the model 5 returns “mul div int+ 1 int+ 2 mul pow x int+ 2 pow x int+ 2” (equivalent to 1/2*(x^2*x^2)).

In other case, given “sub Y' mul INT+ 2 x” (y'= 2x) as input, the model 3 returns “mul pow x int+ 2 pow x int+ 2” (equivalent to x^2*x^2), model 4 returns “mul div int+ 1 int+ 2 pow x int+ 2” (equivalent to 1/2*x^2), and model 5 returns “mul int+ 2 pow x int+ 2” (equivalent to 2*x^2). In this case, model 4 and 5 produce outputs closer to solution.

5.2 Comparison of the Results Between the Original Paper and my Project

The original paper achieved an accuracy of 93% on the full integration dataset, whereas the best model from the project achieved the accuracy of 80% on the partial integration dataset.

The two main differences from the original model is the 1. Dataset size and the 2. Model used. The transformer model would be better at the task of machine translation, especially because mathematical expressions for integration may have long-distance dependencies among input variables. The seq2seq model can lose track of its context sequence, whereas the transformer architecture using self-attention enables the model to focus on the most relevant part of the input sequence at each step and thereby easily transmitting information across the input sequence.

Also, using the full dataset with varied expressions could have improved the accuracy and model generalizability.

5.3 Discussion / Insights Gained

There are four the main take-aways from the project:

Importance of accuracy metric for a task requiring high precision. Given the nature of this task, achieving an accuracy close to 100% is important. Unlike a natural language translation model where “a good enough” translation delivers the meaning of the original input sentence, mathematical tasks require precise output. For example, if integration of “y’=2x” outputs “y=2x^2” instead of “y=x^2”, the model output has a similar form as the solution but is incorrect mathematically. A jumble of expressions that contain the right variables but with different order result in wrong multiplications and additions and return an incorrect solution.

Importance of hyperparameter tuning. The training phase highlights the importance of hyperparameter tuning to improve the model performance.

Importance of appropriate loss function. For sequence generation tasks, setting the right loss function drastically improves the model performance. Before masking off the paddings in the loss function, the model performance suffered and the model accuracy was slow to increase over time. Updating the loss function increased the accuracy from 40% to close to 70% immediately.

6. Conclusion and future work

Goal of the project was to build a machine translation model that converts mathematical expressions into its integrated expression. With hyperparameter tuning and experimenting with different model layers, the model was able to achieve an accuracy of 80% for the integration task. The result differs from the original paper, which achieved 93-95% accuracy using the full dataset. The two hypotheses for the discrepancy is 1. the size of the dataset used and 2. the difference in the model architecture, especially since the lack of

self-attention layer in the seq2seq model may note be able to handle long-term dependencies well.

Still, the moderate achievement in the accuracy through this project suggests a possibility of handling any non-natural language sequence with a pattern that can be fed to the seq2seq model to produce a meaningful output, thereby extending the application of the machine translation beyond natural language.

Key improvements that can be made to the project includes:
1. Providing a function that converts the model output sequence to an equation. Right now with the lack of such a method, the output sequence has to be translated into an equation manually.
2. Using the domain knowledge to complement the model prediction. Ruling out "illegal" expressions in the decoder step using known heuristics can reduce the probability that the model outputs a nonsensical formula. For instance, if the model predicts the next sequence which is clearly invalid (for instance, if the next sequence generated would produce a nonsensical equation such as "1+2 3"), the model can move on to the next most likely token.

7. References

[1] Original paper: Guillaume Lample, Francois Charton. 2019. Deep Learning for Symbolic Mathematics (https://arxiv.org/pdf/1912.01412.pdf)

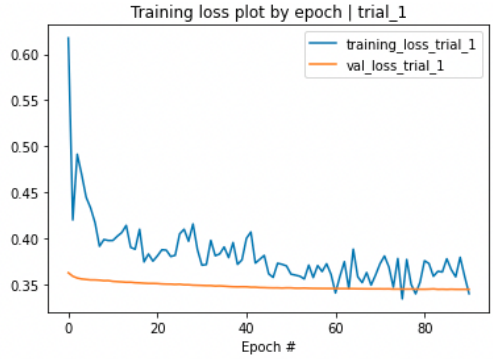
[2] Neural machine translation with attention https://www.tensorflow.org/text/tutorials/nmt_with_attention

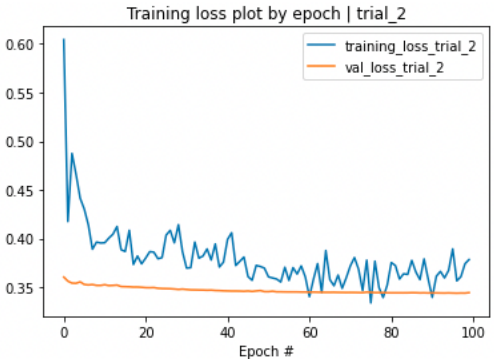
[3] Facebook Research Github page containing dataset for model training (https://github.com/facebookresearch/SymbolicMathematics)

9. Appendix

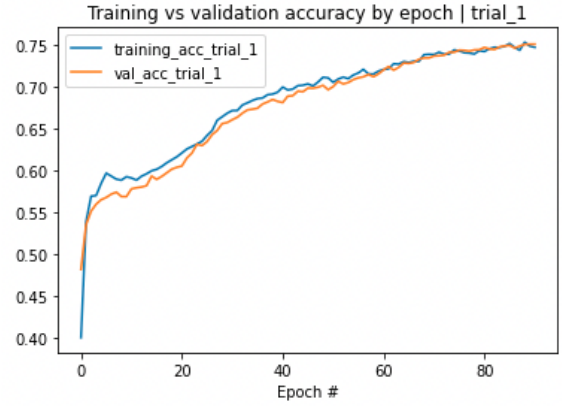
9.1 Loss and accuracy comparison between training and validation phase

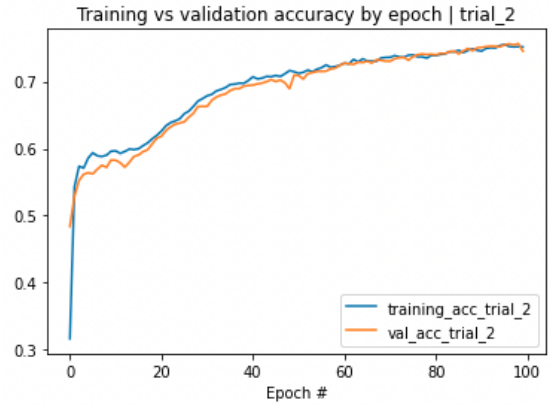
Training vs. validation loss by trial

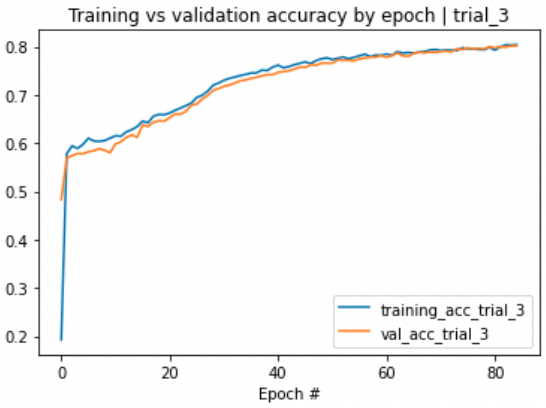


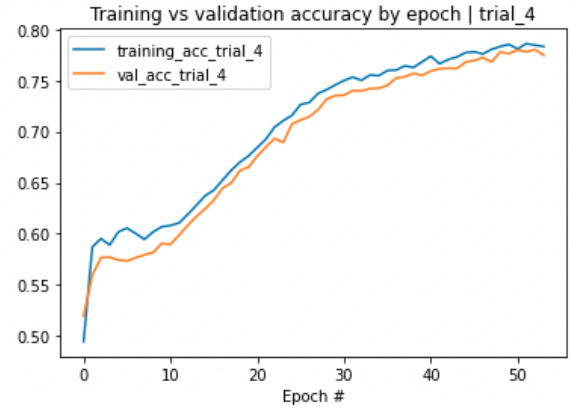


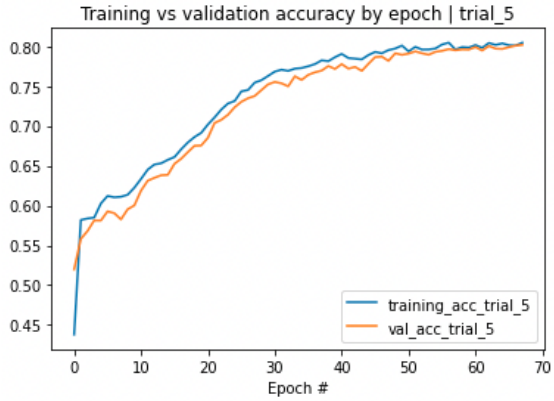
Training vs. validation accuracy by trial






Training vs validation accuracy by epoch | trial_3
training_acc_trial_3
val_acc_trial_3
Epoch #


Training vs validation accuracy by epoch | trial_4
training_acc_trial_4
val_acc_trial_4
Epoch #


Training vs validation accuracy by epoch | trial_5
training_acc_trial_5
val_acc_trial_5
Epoch #