Neural Predictive Calculator
Methodology

Maturitätsarbeit, Kantonsschule Baden
Erstbetreuer: Michael Schneider, Zweitbetreuer: Julia Smits

Anton Mukin

June 2025

Abstract

This study investigates which Neural Network architecture is optimal for predicting simple arithmetic expressions. Various architectures built for regression tasks were evaluated: Feedforward Neural Networks (FNNs), Recurrent Neural Networks (RNNs), transformers, as well as fine-tuned Large Language Models (LLMs). Evaluations were performed to measure if the models were able to learn arithmetic rules by assessing their generalization capabilities with a custom-made benchmark. The best performance on the previously defined benchmark was achieved by the simple FNN architecture.

The good performance of simple architectures with relatively few parameters suggests that these architectures are better at generalizing than the more sophisticated models. Still, no model performed well enough to be classified as *having learned* arithmetic rules. Which leads to the conclusion that neural networks are not capable of extrapolating symbolic rules. Furthermore, the results point to the fact that neural networks differentiate from our human brain in the way they generalize rules from data[1]; While humans typically identify underlying principles, neural networks rely on pattern recognition, primarily analyzing surface-level data.

The regression models used for this project were presented to a colleague in a guided discussion. Although this conclusion is drawn from a single experiment, the results indicate that it was advantageous to explain the basic workings of FNNs and transformers using models for regression as opposed to sequence-to-sequence models.

---

[1]Marcus, 2018 expresses this well in his concerns.

Contents

List of Figures

1 Introduction

The goal of this document is to describe methodology applied in the current project, assist in reproducibility and demonstrate how the findings discussed in the related document were obtained.
All of the code written for this project is available in the following GitHub repository:
https://github.com/AntonStantan/matura

1.1 Brief Summary of the Project

This Matura project investigates and evaluates the arithmetic capabilities of different neural networks.
The project began with a literature review to generate a hypothesis regarding the weaknesses of neural networks in performing simple arithmetic calculations. This literature review was submitted as the Zwischenprodukt, alongside a proof-of-concept notebook featuring a comparison of Feed-forward Neural Networks (FNNs) of different sizes.
The next step was to build a Recurrent Neural Network (RNN) and similar attention-based RNNs to investigate their arithmetic capabilities and compare them to those of the FNN using a benchmark.
The benchmark's baseline was defined as the performance of a basic FNN on different, but broadly similar, arithmetic tasks.
Afterwards, the same procedure was applied to the transformer type of neural network model. Here, their exact functionality was thoroughly investigated because of the unique architectures.
A similar workflow was repeated for larger pre-trained models.
To sum things up, all findings were collected and evaluated comprehensively.

1.2 Software and Hardware Specifications

All code for this project is written in Python notebooks (Jupyter Lab). The preferred library used was tensorflow keras.
Most of the models were trained locally on an Nvidia GPU device: *Nvidia Jetson Orin Nano Super Developer Kit*, or alternatively on a Laptop: *ThinkPad Z16 Gen 1* with a *AMD Ryzen 7 PRO 6850H* CPU, which was used for calculations.

2 Experimental Setup

For most notebooks, the GetXY.py script was used to get the training, test and validation datasets. Other notebooks used the same data, just generated it differently.
For the benchmark, mostly the FNN1_1.py script was used to obtain the performance of the benchmark model.

2.1 Train, Test and Validation - Data

To define the training data, the decision was made to exclusively use subtraction and addition. The arithmetic expressions were defined to consist of two operators (+ or -) and three integers within the range $[-5, 5]$.
The reason for these definitions is that + and - are the two simplest operators. The decision to introduce the model to three numbers, instead of the more common two, was made in the hope of simplifying the transition to more numbers in a later stage.
For example, the first three entries are:

Figure 1: Nvidia Jetson device during its boot sequence.

$$1 - -2 + 3 \qquad -2 + -3 - -5 \qquad 3 - 1 - 0$$

In total, there are 1907 such expressions in the training data.

The test data is also defined in a slightly unconventional manner. It is split into three categories.

- Inside of the number range (1457 expressions): Expressions identical in structure to the training data but not present in the training set.

- Outside of the number range (2048 expressions): The same expression structures, but with numbers in the ranges $[-8, -5]$ and $[5, 8]$, e.g.,

$$-6 + 6 + 8 \qquad -7 + 6 + 7 \qquad 5 - -6 - 6$$

- Longer expressions (700 expressions): Expressions of different lengths. Specifically, there are between 2 and 8 numbers inside of the $[-5, 5]$ range, e.g.,

$$-5 + 1 \qquad 3 + -1 + 4 + -2 + -4 \qquad -2 + 1 + 5 + -5 - -2 + -1 - 2 - 5$$

During training, later on, the model will require a small validation dataset (636 expressions), which will be used to evaluate the model during training on data not in the training dataset. Based on the model's performance on this datset, it will be either rewarded or punished. The validation data is generated the exact same way as the training data, so that there is 3 times as much training data as there is validation data.

2.2 Tokenizer

Because only tensors with floats can be passed into a model, the train and test data are always processed by a tokenizer before entering a neural network, as described by GeeksforGeeks, 2025.
Traditionally, a tokenizer assigns each prevalent character in the data a number, but because this project works with numerical data, this is not needed. Only the characters $+$ and $-$ need to be assigned a number. For this project, the decision was made to work with: 1.0 replacing $+$ and 0.0 replacing $-$.

Padding is also added, because a network trained on short expressions with only 3 numbers cannot process long expressions with more numbers (higher-dimensional vectors). Padding increases the length of those short expressions, by inflating them with values that do not represent any information.
To differentiate it from all other integers, a padding value of 0.5 was defined.

Consequently, an expression that was passed through the tokenizer and extended with padding would appear as follows:
Before tokenizer:
$$1 - -2 + 3$$
After tokenizer:
$$[1. \quad 0. \quad -2. \quad 1. \quad 3 \quad 0.5 \quad 0.5 \quad 0.5 \quad 0.5 \quad 0.5 \quad 0.5 \quad 0.5 \quad 0.5 \quad 0.5 \quad 0.5]$$

2.3 The Benchmark

To calculate the benchmark, a model is evaluated on four categories. Its performance is then compared to that of a baseline model and graded. The four categories are:

- Test data inside the number range defined for training data (like the validation data),

- Test data outside of the number range,

- Test data consisting of longer expressions with numbers inside of the number range,

- A relative MSE of the test data with numbers outside of the range.

In the order previously mentioned, let: `baseline_deviation`, `baseline_out_deviation`... be the baseline values and `meanDiff_InRange`, `meanDiff_OutRange` be the Mean Absolute Errors (MAEs) of the model being evaluated. To make the code for the formula below more apparent, imagine that the benchmark formula establishes a score of 1 for any model that performs identically to the baseline model. Therefore, models that perform worse on the test data will yield scores below 1, whereas models that perform better will result in scores above 1.

```
benchmark = 0
benchmark += baseline_deviation / (meanDiff_InRange**2) / 4
benchmark += baseline_out_deviation / (meanDiff_OutRange**2) / 4
benchmark += baseline_long_deviation / (meanDiff_LongRange**2) / 4
benchmark += baseline_relError / (meanDiff_OutRelRange**2) / 4
print(f"Benchmark: {benchmark}")
```

The use of a relative error penalizes mistakes on "simpler" expressions and is less harsh on more "difficult" ones. Expressions whose absolute result is small are easier for a neural network to solve, while a larger absolute result is more difficult to calculate.

For the same reason, two more adjustments have been made:

· For longer expressions, only expressions with 4 numbers were used.

· For expressions with numbers outside the range, the expression was only used if the absolute values of all three numbers added up to 22.

These subsets were chosen because of their reasonable MSE of around 10 each.

This, as well as the inclusion of the relative MSE, is done in hopes of bringing the benchmarks of different models closer together and increasing the linearity between them. This makes the benchmark more tractable and consistent for comparison.

The baseline values are defined to be from an FNN model[2] with 30 neurons and two dense layers. It is trained over a period of 200 epochs with early stopping enabled.

3 Feed-forward Neural Networks (FNNs)

For details on how an FNNs works, please refer to section 2 in the literature study.
This section corresponds to the /FNN directory in the GitHub repository.
There are three notebooks here: FNN1, FNN2 and FNN3.

3.1 Training a Neural Network Using Tensorflow

This subsection will show how the models used in this project were first defined and then trained with TensorFlow on the example of a simple FNN.

To begin, the necessary libraries are imported.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Dense, Layer, Dropout
from tensorflow.keras import layers, Sequential
from tensorflow.keras.layers import PReLU
```

Next, a dataset is created from the train and validation data discussed in the previous subsection 2.1. Batches of 32 are used. The validation data is similar to the in-range test data, but with fewer expressions.

```
batch_size = 32
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
.shuffle(len(x_train)).batch(batch_size)
val_dataset = tf.data.Dataset.from_tensor_slices((x_val, y_val))
.batch(batch_size)
```

---

[2] The number of neurons and layers was chosen, so that the model is capable of predicting with a reasonably good performance on the validation data. (This was done in FNN1.ipynb.) The exact number of neurons and layers does not really matter, because all models whose benchmark is being evaluated are compared to the same baseline model.

The model can now be defined using tf.keras.Sequential. The model's architecture is clearly visible. In this example, the model has two dense layers with 64 neurons each.

The activation function used is PReLU, as this was the most promising in Trask et al., 2018.

Dropout is included to prevent overfitting. In most models used in this project, it was not necessary.

As shown below, the input shape must be defined when defining the model.

Additionally, it should be noted that the output layer only consists of a single neuron with a linear activation function. This means the model created is a regression model, which is similar to not having a decoder.

```
input_shape = (15,)
model = Sequential([
    keras.Input(shape = input_shape),     #input

    Dense(64),                            #first dense layer
    PReLU(),                              #PReLU activation function
    Dropout(0.1),                         #dropout layer

    Dense(64),                            #second dense layer
    PReLU(),
    Dropout(0.1),

    Dense(1, activation='linear')         #output layer
])
```

The next step is to compile the model by choosing an optimizer and a loss calculation, in this case, Mean Squared Error (MSE).

```
model.compile(optimizer="adam", loss="mse")
```

The last remaining step is to fit the model on some data. The model will be fitted on the previously defined train and validation data. The training process will take 200 epochs to complete or might be aborted preemptively if the model starts to overfit and early stopping is triggered.

```
model.fit(
    train_dataset,
    validation_data=val_dataset,
    epochs=200,
    callbacks=[early_stopping],
    verbose=1
)
```

3.2   FNN1 and FNN2 Notebooks

In FNN1, a simple FNN was created, its performance was evaluated, and FNNs of different sizes were compared against each other in a heatmap. It is noteworthy that the models here were used with a bootstrap, meaning multiple models with the same sizes were trained and then used to give one combined prediction. This was done to reduce noise. In later notebooks, this will not be the case, as this makes it more difficult to accurately evaluate a model.

FNN2 includes a model with hyperparameters (in this case, the number of neurons, the number of layers, and whether or not to use dropout) chosen by the KerasTuner. This is, in simple terms, an automation process: it picks out models with different hyperparameters, trains them, and compares their performance on validation data. The model with the best-performing hyperparameters will be chosen as the best model. The model in FNN2 is evaluated inside that notebook, and its benchmark is also calculated there.

3.3   FNN3 With Positional Encoding

In the notebook FNN3, a positional embedding layer is added to the model, and a Keras tuner tunes the hyperparameters of the model to an optimum. The formula used is the one from Vaswani et al., 2023.

4 Recurrent Neural Network (RNN)

RNNs process sequential data by calculating a hidden state value, which is passed from one timestep to the next. For details on how RNNs work, refer to the corresponding section in the findings document.

4.1 RNN0 and RNN2

The RNN0 notebook is a simplistic model with RNN architecture.
An RNN model consisting of 2 dense layers with 50 neurons each:

```
model = keras.Sequential([
    keras.Input(shape=(input_shape, 1)),
    keras.layers.SimpleRNN(50, return_sequences=True),
    keras.layers.PReLU(),
    keras.layers.SimpleRNN(50),
    keras.layers.PReLU(),
    keras.layers.Dense(1, activation = "linear")
])
```

Note that 'return_sequences = True'. This is necessary for a subsequent layer. By default, this is set to false because RNNs are frequently used with just one layer for applications like Natural Language Processing (NLP) or time-series analysis, such as stock market prediction.
The same notebook used in FNN2 was adapted to work with RNNs in the notebook RNN2. As in FNN2, the KerasTuner was used for finding the optimal number of neurons and layers, as well as whether or not to include dropout after a dense layer.
Useful sources for the creation of the first RNN prototype: Bowman et al., 2015; Stryker, n.d.; Zhu and Chollet, 2023

5 Attention and Transformers

5.1 Attentional RNNs

For the sake of transitioning from RNNs to transformers, an attentional RNN model was trained and evaluated.
It consisted of a bidirectional[3] Long Short-Term Memory (LSTM) layer, used as the encoder, and a self-attention mechanism for attention.

```
#Encoder:
encoder_inputs = Input(shape = (len(x_train[0]), 1))
encoder_outputs = Bidirectional(LSTM(64, return_sequences=True))(encoder_inputs)

#self-attention mechanism
attention_outputs = Attention()([encoder_outputs, encoder_outputs])

#condensing into a single vector
context_vector = GlobalAveragePooling1D()(attention_outputs)

#output layer (Decoder)
output = Dense(1, activation = "linear")(context_vector)

model = Model(inputs = encoder_inputs, outputs = output)
```

This bidirectional LSTM with attention was realized in the g4gLSTM notebook. It contains a model built with the help of code from "Adding Attention Layer to a Bi-LSTM", 2025.

---

[3]This means input is being processed from front to back and from back to front. It allows the LSTM to get a richer and broader contextual representation.

A number of 35 LSTM units were chosen for the encoder because of its previous performance (with boot-strapping), visible in a heatmap.

In the directory /RNN/Heatmaps, some heatmaps are presented; these were used to evaluate which model architecture to add attention to. Even though Gated Recurrent Units (GRUs) showed better results, they were not implemented because of the lack of online documentation for using attentional GRUs. This led to the choice of attentional LSTMs.

Nevertheless, the GRU architecture is still discussed in the findings document.

5.2   Transformers

The original paper by Vaswani et al., 2023 was used as an example for building a replica model, which later was adjusted for better performance on the specific tasks in this project. The architectures used in this project resemble the one that first introduced the transformer architecture with one small difference: They are Regression models[4], not the sequence-to-sequence (seq2seq) type from the paper. Meaning they use an output layer consisting of just one neuron[5]. It linearly projects the matrix passed through the model down to a single number, to obtain the output value (prediction).
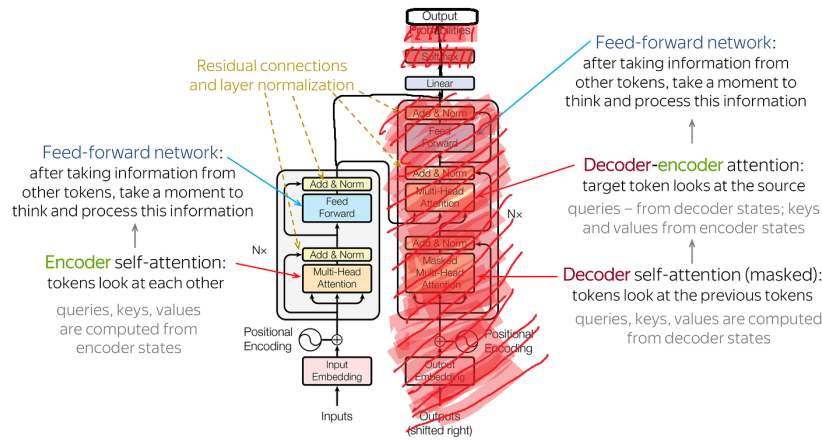
Figure 2: A Seq2Seq model from Vaswani et al., 2023. The Encoder-only model used in this project is the same, except the decoder part is skipped; in the image, this part has been crossed out with red. Source for figure

To code this in Python, an object-oriented approach has been taken, with classes consisting of other classes in a nested structure, similar to a matryoshka doll.

Below is the hierarchy:

- Transformer which encapsulates embedding and positional encoding, as well as the encoder and a final one-neuron-FNN with a linear activation

- Encoder containing encoding layers and dropout

- Encoding layer consisting of the MHA and pointwise-FNN, as well as layer normalization and dropout (included in the original paper to prevent dropout)

- Multi-Head Attention (MHA) and pointwise-FNN

- Scaled Dot-Product (SDP) Attention

---

[4]The models used have a similar Encoder-only architecture, like the BERT architecture proposed by Devlin et al., 2019.

[5]In a seq2seq model the Decoder would be responsible for printing the output sequence, which is not necessary for this project.

A model with the same hyperparameters as the ones used by Vaswani et al., 2023 can be found in transformer0. Unfortunately, this model gets stuck in a local minimum during training, as can be seen at the bottom of its notebook. The minimum is simply predicting a number close to 0 for every expression.

This is an undesirable outcome, necessitating the tuning of the model's hyperparameters. This is done with the help of KerasTuner.

A couple of additional minor adjustments were made to the model before applying the KerasTuner, mainly switching to the AdamW optimizer, because it is better than Adam for transformers according to Loshchilov and Hutter, 2019. A learning rate scheduler with warmup and cosine decay was also used. This means that, unlike before when the learning rate stayed constant throughout the training, the learning rate now changes. It is higher in the beginning and decays towards the end. This assists the model in taking larger optimization steps initially, thereby overcoming local minima, before proceeding with more precise, smaller steps towards the end of the training.
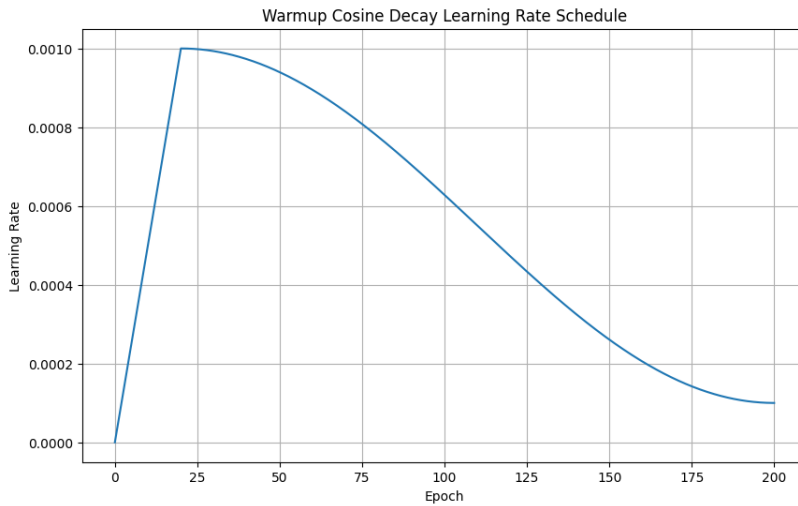
Figure 3: This is how the learning rate could look like for a model being trained over a period of 200 epochs when using a cosine decay with a linear warmup, as in this project. Except for the changing peak learning rate, all other parameters are the same as the ones used in this project.

In contrast to the FNN or the RNN, a transformer has to be trained on other hyperparameters:

- The number of heads inside the MHA

- The number of dimensions vectors have inside the model

- The number of encoding layers

- The number of neurons in the layer inside of the pointwise-FNN

- Whether to use dropout or not

- The peak learning rate after warmup and after the cosine decay

In the notebooks transformer4 and transformer5, you can look at models with successfully tuned hyperparameters displaying promising performance on the benchmark.

The difference between them is that Transformer5 is a larger model than Transformer4.

6 Pre-Trained Transformers

The final model type to be analyzed was fine-tuned pre-trained LLMs. Since working with them is completely different than working with locally trained neural networks, extensive research was required before commencing the work.

Significant effort was dedicated to resolving library compatibility issues for the amd64 architecture system on the Jetson device. The final solution was using jetson-containers, which offer a Docker container optimized for the Jetson system. Specifically, the container called 'bitsandbytes' has most of the libraries needed for this project; the few remaining ones can be installed manually using: `pip install`.

6.1 Gemini 2.5 Pro

The first model that was fine-tuned was Gemini 2.5 Pro. This model was selected partly because of the author's familiarity with it and its high ranking in the LMArena. Fine-tuning had to be done on a remote server hosted by Google Cloud because of the model's size.
In the end, the costs for their service amounted to CHF 83.58.
The fine-tuning was done using VertexAI services. First, a .json file with the training data[6] had to be created and uploaded. Afterwards, the model was fine-tuned in the notebook gemini_vertex using Vertex SDK. The resulting model is loaded into an endpoint on the Google Cloud servers.
To access the new, fine-tuned model and be able to evaluate it, a fairly simplistic notebook is created: gemini_vertex_predict.
There, the model is loaded in from its endpoint and used to predict test data.

6.2 Gemma 3

For diversity, a small (Gemma3 1B parameters by Team, 2025a) and a tiny (Gemma3 270M parameters by Team, 2025b) model were chosen. They have been provided by Google on Huggingface.
Because of their size, they can be fine-tuned locally on the Jetson device. They are fine-tuned with the SFTTrainer provided by 'trl'.[7] The guide on Hugging Face was used as help for this.
During the fine-tuning process, a custom-designed function calculates the accuracy on some validation data (defined here as the first 100 samples from the test data). This function was used to evaluate the model at different steps in training.
Fine-tuning notebooks for Gemma3 models can be found under: Gemma3 1B, Gemma3 270M

After fine-tuning, the models can be found in the Hugging Face repositories:
fine-tuned Gemma3 1B, fine-tuned Gemma3 270M
After a base model of the same architecture is loaded with the parameters obtained from the fine-tune, see the notebooks:
Gemma3 1B prediction notebook, Gemma3 270M prediction notebook
There, the models are used to predict test data and, again, calculate the accuracy of the respective model.

6.3 Anomalous Results Encountered After Evaluation

An anomalous observation was made: The accuracy during the fine-tuning process is greater than the accuracy calculated when evaluating the models in the prediction notebooks. This is likely due to an overlooked error, though none were found upon investigation.
The discrepancy was approached systematically; multiple hypotheses were formulated and subsequently disproven. The source of the error, if one exists, could not be identified.

Hypotheses for the Cause of the Discrepancy:

- **Incorrect Model Loading:** The first hypothesis was that a previous or incorrect model checkpoint was being loaded, instead of the most recent one.
  - *Result:* All previous runs were deleted, leaving only one. The specific, most recent checkpoint was explicitly specified.
- **Data Generation Error:** The second hypothesis suggested an error in the data generation process, such as using different hyperparameters (e.g., temperature) between training and evaluation.

---

[6]For fine-tuning, the training set used was the combination of the training and validation data from before. This is because the VertexAI website had a tedious setup for validation data and this would not have a big impact on results anyway.
[7]We cannot use the standard Trainer from Hugging Face because the fine-tuning we do is supervised; hence the name: Supervised Fine-Tuning (SFT)

- – *Result:* Both the PyTorch '.generate()' method and the Hugging Face generator pipeline yielded similar results. The pipeline uses hyperparameters from the fine-tuning configuration.
- **Error in Evaluation Calculation:** Another possibility was an issue with the calculation logic within the 'compute_metrics' function during the evaluation steps.
  - – *Result:* The function was thoroughly checked to ensure the validation dataset was used, not the training dataset. The function's logic is straightforward and was verified to be correct.
- **Model Overfitting:** The model might be overfitted to the training data.
  - – *Result:* Definitely not, this does not align with the definition of overfitting, but just to be sure; Predicting on the training data yielded nearly identical (or slightly better) results to those calculated from test data, which does not support overfitting as the cause of the discrepancy.
- **Calculation Discrepancy:** There might be a subtle difference in how predictions are processed. In the 'compute_metrics' function, the prediction is only counted if the model uses a turn separator. This could explain the gap in accuracies.
  - – *Result:* After rerunning the training with this correction, the accuracy dropped slightly from 100% to 97% for the Gemma 3 270M model, but a significant gap still remained.
- **Evaluation Mode ('eval()'):** The pre-trained model contains dropout and layer normalization. If the model is not set to evaluation mode ('model.eval()'), these layers would behave incorrectly during inference, potentially worsening performance.
  - – *Result:* This was not the case. After setting 'model.eval()', the accuracy for the Gemma3 270M model only increased by approximately 0.5%. 'model.eval()' did not have an effect.
- **Improper Weight Loading:** The final explanation could be that the model's weights are not being loaded into the model properly. This was the case when working on FNN2 once.
  - – *Result:* This is unlikely, as loading hyperparameters is a different process from loading trained weights. A model with loaded weights is already trained. Furthermore, the model shows clear signs of having learned from the fine-tuning process (e.g., in its response format).

Should a reader identify the cause of this discrepancy, they are kindly encouraged to contact the author of this project.

7 Findings Table

The findings table in the Findings section of the Findings document presents a comprehensive summary of all model performances. A table is used to present all the important regression-based-models and their performances.

FNN2, FNN3, RNN2, Bidirectional LSTM with attention, transformer4, as well as transformer5 were the chosen models. They were chosen so that all major different architectures were covered. The strongest performing models[8] were chosen for each architecture. All of them were evaluated on:

- Mean Absolute Error (MAE) on all three of the test datasets separately

- Mean Relative Error (MRE) on the test datasets, excluding the one with longer expressions

- The Benchmark score

- p-value of the Benchmark test

Each model was trained 5 times separately on the same training data, and then evaluated on the same test data. The average performance of the 5 evaluations was used in the table.

7.1 p-Value

The p-value was calculated using the scipy-stats library.

The p-values used are two-sided one-sample t-test[9]as first described by Student, 1908. They were calculated using the list of 5 benchmarks computed in the 5 trainings performed for each model.

The null hypothesis is defined as benchmark value $= 1$, the performance of the baseline model.

The p-value is a value between $[0, 1]$. It represents the probability of observing these benchmark results or more extreme ones, if the model being evaluated performs the same as the baseline model.

---

[8]The best performing models were determined with Keras-Tuner.

[9]A t-test was chosen over a gaussian normality curve because of the small sample size as suggested by de Winter, 2013.

```
from scipy.stats import ttest_1samp
import math

# Values taken from the notebooks, where the p-value was calculated slightly inaccurately.
benchmark_list = ['10.600075', '12.701344', '10.435802', '10.592991', '9.183982'] #FNN2 values
numeric_values = [float(i) for i in benchmark_list]

log_transform = [math.log(x) for x in numeric_values]
print(log_transform)

stats, p_value = ttest_1samp(log_transform, popmean = 0)

print(f"p-value of the benchmarks: {p_value}")
```

In the notebooks, the p-value was computed with a slight inaccuracy (it also accounted for negative benchmarks which are not possible by definition.) The p-value calculation script (seen above) calculates the p-values correctly by applying a logarithmic transformation on the individual benchmarks first. This is necessary because the benchmark can only be positive by defintion and a two-sided t-test also accounts for negative numbers. The logarithmic transformation converts values between $(0, 1)$ to be negative. It also changes baseline benchmark: $ln(1) = 0$

7.2 Fine-Tuned Models

In addition to regression-based architectures, a table was also drawn for fine-tuned language models.
The performance of the Small and Large Language Models was evaluated with accuracy, not deviation, because these types of models generate discrete tokens for each number in an expression. For them, solving an expression isn't a regression task, but a classification task. This means the output of these models will always be an integer, never a float like with the regression models. Compared to the solution of an expression the model's output will either be right or wrong.

While the Gemma3 models are open-sourced and their parameter sizes published, the Gemini 2.5 Pro and its parameters on the other hand aren't open-sourced to the public. An estimation for the number of parameters of the Gemini 2.5 Pro model was published by Chadha, 2025. This estimate was adopted for the table.

8 Guided Discussion

After having evaluated all data collected on the performance of various models, a guided discussion with a colleague was organized. The goal was to test a hypothesis formed at the very start of the project: **It is simpler to explain how models for regression (like the ones used for this project) work, than the usual sequence-to-sequence (seq2seq) models.**
The discussion partner was selected because he fit the criteria for the target audience of this project: Basic prior knowledge in both the theory, as well as the application and implementation of neural networks.
The discussion was structured as an open presentation where the author explained the functionalities of FNN and transformer regression models with a focus on their architecture. To support the explanation, clear examples from the main project were used to illustrate the point.
After the approximately one hour long discussion, feedback was collected through a questionnaire, which also contained a quiz. The goal was to see if regression models helped the participant understand the new topics, as well as evaluate his understanding.
The results of the questionnaire along with the author's comments are available on github.

9   Closing Remark

The author has invested significant effort into this project and would appreciate any feedback. He can be reached via email at anton.mukin@students.ksba.ch or lolgod2703@gmail.com.
If you find this project interesting, please consider starring the GitHub repository.

The author would like to thank Herr Schneider for his valuable insights and prompt responses.

References

Adding attention layer to a Bi-LSTM [Accessed: Oct. 09, 2025]. (2025). https://www.geeksforgeeks.org/nlp/adding-attention-layer-to-a-bi-lstm

Bowman, S. R., Potts, C., & Manning, C. D. (2015). Recursive neural networks can learn logical semantics. https://arxiv.org/abs/1406.1827

Chadha, A. (2025, March). Gemini 2.5: Google's revolutionary leap in ai architecture, performance, and vision. Retrieved November 8, 2025, from https://ashishchadha11944.medium.com/gemini-2-5-googles-revolutionary-leap-in-ai-architecture-performance-and-vision-c76afc4d6a06

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. https://arxiv.org/abs/1810.04805

de Winter, J. C. (2013). The student's t-test and the one-way analysis of variance: A comparison using small sample sizes. *Practical Assessment, Research, and Evaluation*, *18*(1), 10.

GeeksforGeeks. (2025). Nlp — how tokenizing text, sentence, words works. https://www.geeksforgeeks.org/nlp/nlp-how-tokenizing-text-sentence-words-works

Loshchilov, I., & Hutter, F. (2019). Decoupled weight decay regularization. https://arxiv.org/abs/1711.05101

Marcus, G. (2018). Deep learning: A critical appraisal. https://arxiv.org/abs/1801.00631

Stryker, C. (n.d.). What is a recurrent neural network (rnn?) [Accessed: Oct. 09, 2025]. https://www.ibm.com/think/topics/recurrent-neural-networks

Student. (1908). The probable error of a mean. *Biometrika*, *6*(1), 1–25. Retrieved November 9, 2025, from http://www.jstor.org/stable/2331554

Team, G. (2025a). Gemma 3. https://goo.gle/Gemma3Report

Team, G. (2025b). Gemma 3. https://arxiv.org/abs/2503.19786

Trask, A., Hill, F., Reed, S., Rae, J., Dyer, C., & Blunsom, P. (2018). Neural arithmetic logic units. https://arxiv.org/abs/1808.00508

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023). Attention is all you need. https://arxiv.org/abs/1706.03762

Zhu, S., & Chollet, F. (2023). Working with RNNs [Accessed: Oct. 09, 2025]. https://www.tensorflow.org/guide/keras/working_with_rnns