Neural Predictive Calculator

Findings

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

This project began with a simple idea: a calculator that predicts answers using a neural network rather than performing systematic calculations. The central question that propelled this project was determining the most suitable neural network architecture for this task. The current document presents the findings from the "Neural Predictive Calculator" project.

1.1 Hypothesis

Following a literature review, the expected results were as follows: It was hypothesized that Feed-forward Neural Networks (FNNs) would be the weakest architecture, Recurrent Neural Networks (RNNs) would perform better, and transformers and pre-trained transformers would perform the best. Technologies such as positional encoding, a seq2grid pre-processor[2] and a PReLU activation function were expected to help the model generalize simple arithmetic rules.

2 Feed-forward Neural Networks (FNNs)

The functionality of FNNs has been previously discussed in the literature review and methodology document. It will not be further discussed in here. However, FNNs will be referenced later in this document.

2.1 Backpropagation

2.2 Hyper-Parameter Tuning

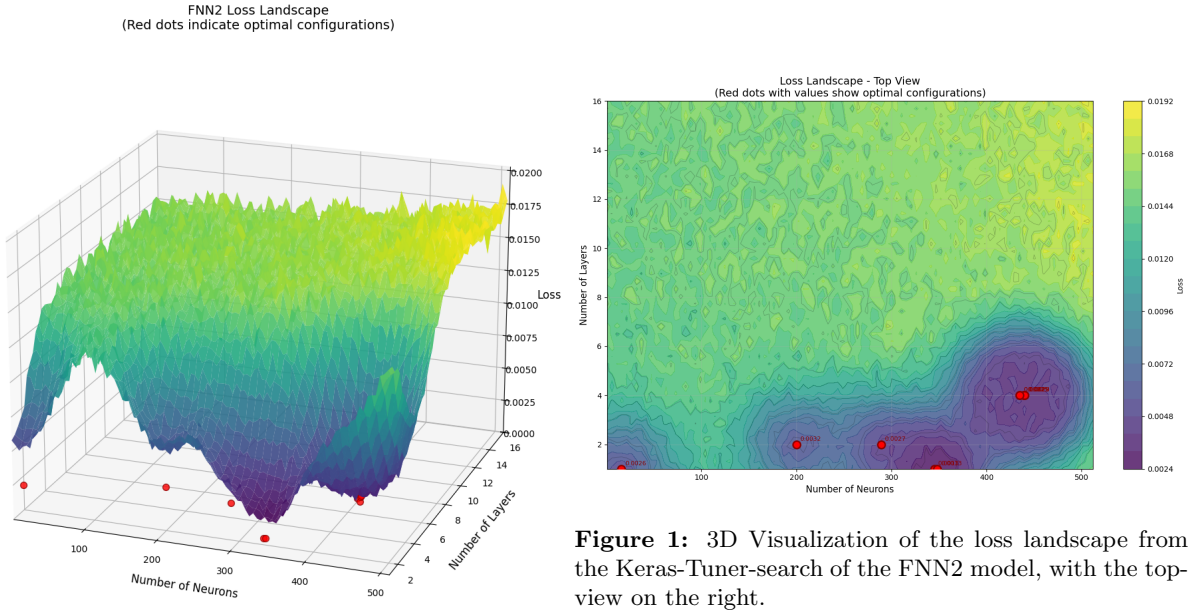

Figure 1: 3D Visualization of the loss landscape from the Keras-Tuner-search of the FNN2 model, with the top-view on the right.

FNNs as well as other models have hyper-parameters (i.e. the number of layers or neurons per layer) which have to be optimized for the specific task the models are being trained on. In this project the Keras-Tuner (or a heatmap visualization of different models) is used to find the global minimas of the loss landscape[3], as

[2]Upon further research into the preprocessor, its architecture uses a neural network for reshaping inputs into a grid, as shown in their paper Kim et al., 2021. This does not fit the requirements for this project. Even though it is not systematic, the preprocessor effectively just increases the model complexity by attaching a RNN in the front. Neural Networks using a seq2grid preprocessor were not evaluated in this project.

[3]The code for drawing the landscape was generated with AI. The data fed into the drawing is real data from the FNN2 notebook. Random noise is added to the landscape to make it look more realistic.

shown below.

3 RNN

Recurrent Neural Networks (RNNs) work similarly to FNNs with one key difference: There is a vector called the hidden-state. This vector contains information about previous time-steps. The hidden-state of the previous time-step, in addition to the input of the current time-step, is fed into a model which computes the hidden-state of the present time-step. The output of each time-step is calculated by feeding the respective hidden-state to a model.

3.1 Numerical Visualization of a RNN:

Let:

- x_t: input at time step t
- h_t: hidden-state at time step t
- y_t: output at time step t
- W_{xh}: weight matrix connecting input to hidden-state
- W_{hh}: weight matrix connecting previous hidden-state to current hidden-state (recurrent weights)
- W_{hy}: weight matrix connecting hidden-state to output
- b_h: bias vector for the hidden layer
- b_y: bias vector for the output layer
- σ: activation function (in our case: PReLU)
- σ_{out}: activation function for the output (linear for the regression task in this project)

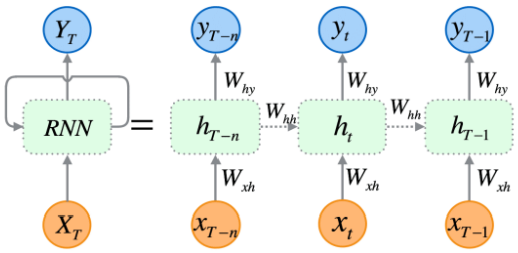

Figure 2: RNN diagram as described by Gawde, 2021

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$
$$y_t = \sigma_{out}(W_{hy}h_t + b_y)$$

Source for equations: Amidi and Amidi, 2019

3.2 Relevant Takeaway

For this project, it means the RNN processes an expression sequentially, one part at a time, rather than as a whole. Due to the nature of the RNN's formula, tokens that appear later in a sequence have a more significant impact on the model's prediction than earlier tokens. This means the output number will almost always be closer to the last number of the expression than the first. This is a common issue with RNNs. It was well documented by Pascanu et al., 2013 and is widely known as the vanishing gradient problem.

4 Other Types of RNNs

To address the vanishing gradient problem, several architectures have been developed, for example, the Long Short-Term Memory (LSTM) proposed by Hochreiter and Schmidhuber, 1997, the Gated Recurrent Unit (GRU) proposed by Cho et al., 2014 or later, the concept of attention proposed by Bahdanau et al., 2016.

4.1 Long Short-Term Memory (LSTM)

The LSTM architecture solves the gradient vanishing problem by using a memory cell. The model can use this to store (5), forget (1) and pass information from the memory cell to the hidden-state (6).
Let:

- $\sigma(\cdot)$ denotes the sigmoid activation function,
- $\tanh(\cdot)$ is the hyperbolic tangent function,
- $\odot$ represents element-wise (Hadamard) multiplication,
- $[h_{t-1}, x_t]$ is the concatenation of the previous hidden-state h_{t-1} and the current input x_t,
- W_f, W_i, W_C, W_o are trainable weight matrices,
- b_f, b_i, b_C, b_o are trainable bias vectors,
- C_t is the current memory cell state,
- $\tilde{C}_t$ is the candidate cell state,

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{1}$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2}$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{3}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{4}$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{5}$$
$$h_t = o_t \odot \tanh(C_t) \tag{6}$$

Source for the equations: GeeksforGeeks, 2025a

In the equations one can see the forget gate activation (1), the input gate activation (2), the candidate cell state (3) and the output gate activation (4).
The cell state is calculated in (5). There, the forget gate which scales the previous cell state is combined with the input gate.
The hidden-state is calculated in (6), where the output activation is applied to the cell state.

4.2 Gated Recurrent Unit (GRU)

The GRU architecture works in a similar way to the LSTM. Instead of utilizing memory cells, GRUs directly use the hidden-state.
Let:

- $\sigma(\cdot)$ is the sigmoid activation function,
- $\tanh(\cdot)$ is the hyperbolic tangent function,
- $\odot$ denotes element-wise (Hadamard) multiplication,
- $[h_{t-1}, x_t]$ is the concatenation of the previous hidden-state and current input,
- W_z, W_r, W_h are trainable weight matrices,
- b_z, b_r, b_h are trainable bias vectors,
- $\tilde{h}_t$ is the candidate hidden-state

- h_t is the current hidden-state

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z) \tag{7}$$
$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r) \tag{8}$$
$$\tilde{h}_t = \tanh(W_h[r_t \odot h_{t-1}, x_t] + b_h) \tag{9}$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{10}$$

Source for the equations: GeeksforGeeks, 2025b

Above you can see the update gate activation (7), as well as the reset gate activation (8).
Further down, the reset gate activation is applied to the previous hidden-state to calculate the candidate hidden-state (9).
Lastly the hidden-state can be calculated by applying (1 - the update gate activation) to the previous hidden-state and combining it with the update gate activation applied to the hidden-state candidate (10). Depending on whether the update gate activation is larger or smaller, the previous hidden-state or the candidate hidden-state will weigh in more on the current hidden-state.

4.3 Bidirectional LSTM with Attention

An architecture of this type consists in part of a bidirectional LSTM, meaning two LSTMs working in parallel, one of which processes data from front to back, the other from back to front; their results are then concatenated and passed on.

The other part of this architecture is the attention mechanism. Here it is, as described by Bahdanau et al., 2016.
Let:

- $\mathbf{h}_t \in \mathbb{R}^{128}$ are the hidden-states for each timestep t, the output from the bidirectional LSTM,
- $\mathbf{W} \in \mathbb{R}^{128\times128}$ is a trainable weight matrix,
- $\mathbf{b} \in \mathbb{R}^{128}$ is a bias vector,
- $\mathbf{u} \in \mathbb{R}^{128}$ is a trainable context vector.

For each time step t, compute:

$$\mathbf{v}_t = \tanh(\mathbf{W}\mathbf{h}_t + \mathbf{b}) \tag{11}$$
$$e_t = \mathbf{u}^\top \mathbf{v}_t \tag{12}$$

Normalize scores using softmax:

$$\alpha_t = \frac{\exp(e_t)}{\sum_{j=1}^{T} \exp(e_j)} \tag{13}$$

The attention weights $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_T]$ indicate the importance of each time step.
Then compute the context vector:

$$\mathbf{c} = \sum_{t=1}^{T} \alpha_t \mathbf{h}_t \tag{14}$$

Source for equations: Cristina, 2023

An attention score is calculated for each timestep in (11), (12) and it is then normalized in (13). Finally the attention weights scale their respective hidden-states from the LSTMs, to compute the context vector (14).

5 Transformers

As of the most recent findings by Zhao et al., 2025, transformer architectures remain the state-of-the-art and most prevalent models across a majority of tasks; they form the basis for LLMs. The key to their success is multi-head self-attention.

In this document, we will discuss the transformer architecture as first proposed by Vaswani et al., 2023.

5.1 Multi-Head Self-Attention

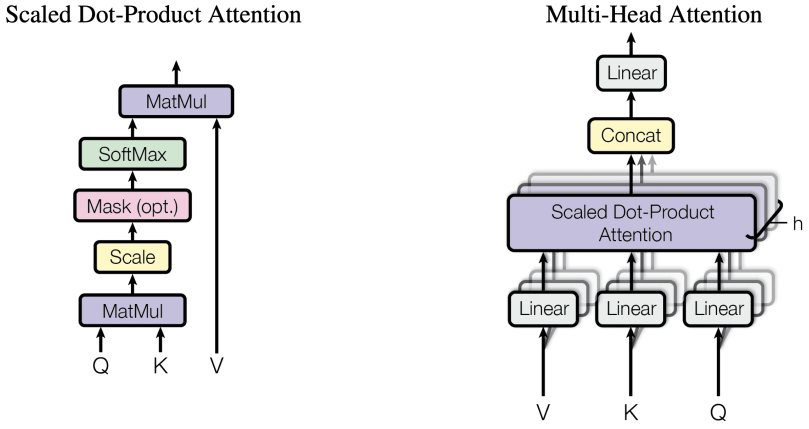

Figure 3: Two diagrams from Vaswani et al., 2023 with the scaled dot-product attention described on the left and multi-head attention on the right.

Data is first split equally into multiple heads, which process it in parallel. There, the tensors are linearly projected with trainable weights to obtain queries (Q), keys (K), and values (V), which can be seen at the bottom of the image.

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

Afterwards, the dot-product between the queries and the keys is calculated and scaled[4] to determine how similar they are. This leaves us with a number between 0 and 1, representing how much attention to pay. The value V represents the actual information of the token for which we just calculated the attention. This means our final step is to scale the value V by its attention. Pay attention to the equation below.[5] (15)

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{15}$$

$$\text{where}^{[6]}\text{: } \text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K$$

This is done across all heads in parallel. The results are then concatenated[7] back together and they undergo a linear projection. As described below:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$
$$\text{where: } \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

[4] According to Vaswani et al., 2023 this is done to counteract dot-products growing too large and later overwhelming the softmax function.

[5] K has a T superscript, this means the matrix is transposed. This is necessary for multiplication because the Q and K matrices have the same number of rows and columns.

[6] Effectively the softmax function sets the biggest element in a set to 1 and the smallest element in the set to 0. All elements in between are scaled appropriately so all elements add up to 1.

[7] This means they are reassembled to have the same dimensions as before they were split up.

5.2 The Encoder Layer

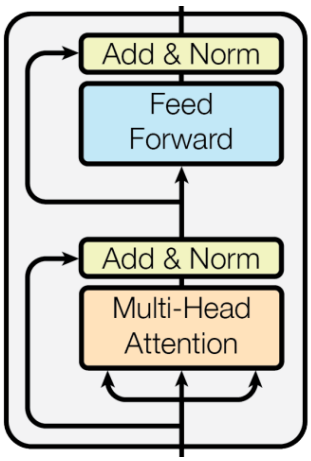

Figure 4: Encoder Layer as described by Vaswani et al., 2023

In figure 4 the multi-head attention first undergoes a residual connection as well as a layer normalization[8], as described below:

where Sublayer(x) is the function, the residual connection is formed by:

$$\text{Output}(x) = \text{LayerNorm}\big(x + \text{Sublayer}(x)\big) \tag{16}$$

Afterwards, all tokens are fed into a point-wise FNN, where they are processed separately and independently. This part is crucial, to introduce non-linearity into the system, as multi-head attention is linear, and non-linearity is needed for a model to be able to learn.

5.3 Point-wise Feed-Forward Network

Equation (17) and figure 5: The pointwise FNN works by taking in a number of d_model tokens, then a layer with a number of d_FNN neurons with weights and biases W_1 and b_1 is applied to them individually. The activation function ReLU discards all negative values and the output layer with weights and biases W_2 and b_2 resets the data back to its original dimensionality.
Please turn your attention to figure 4. After the resulting residual connection from the MHAttention (16) is processed by a pointwise FNN, its residual connection will be the output of a single Encoder Layer as shown in figure 4.
A regressive transformer model, like the one used in this project, consists of multiple such encoding layers and last but not least a single neuron, which works as the output layer.

$$\text{pointwiseFNN}(x) = \max\big(0,\, xW_1 + b_1\big)W_2 + b_2 \tag{17}$$

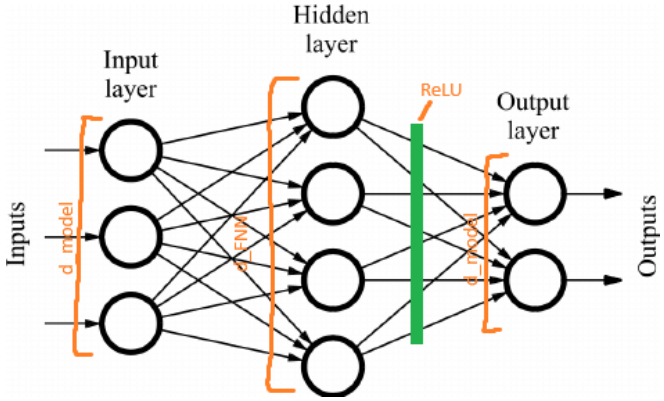

Figure 5: A diagram from Emil, 2020 representing the structure of a pointwise FNN.

[8]Given an input vector, layer normalization works by normalizing its features and making it so their mean is 0.

5.4 Positional Encoding

A special characteristic of the transformer is that data is passed through it as a whole. This means the model does not have positional understanding on its own. To counteract this effect, a positional encoding is applied to the input sequences after the tokenizer.
Let:

- d_{model}: dimensionality of the sequences passed to the model
- pos: index of the token inside the sequence
- i an integer that indexes half of the model's dimensions

Formulas as described by Vaswani et al., 2023:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \tag{18}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \tag{19}$$

This project implements positional encoding as described by Vaswani et al., 2023. Very vaguely put, it assigns a number to each token, based on its position in the sequence.

6 Fine-Tuned Pre-Trained LLMs

The last model type used for this project are fine-tuned Large Language Models (LLMs). For sources and further reading on this section see (in chronological order; left to right): Bergmann, n.d.; GeeksforGeeks, 2024; Srinivasan Anusha, 2024; Stryker, n.d.

6.1 LLM

LLMs adopt the transformer architecture[9] and adapt it to imense sizes, by increasing the architecture's hyperparameters, as well as employing new technological advancements. The number of parameters used by a LLM varies from model to model, but lies in the hundreds of billions per popular high-end LLM. The large size allows large architectures to excel at complicated tasks like mostly Natural Language Processing (NLP). To facilitate training and prediction on such large architectures, massive supercomputers are used, consisting of thousands of GPUs. They mainly work by utilizing parallel processing to distribute the load across multiple powerful GPUs. While by far not a supercomputer, a tiny replica was used for this project: The Nvidia Jetson Orin Nano Super Developer Kit.

6.2 Fine-Tuning Process

Most of the time LLMs are trained on huge dumps of unlabeled data. For more details on this view Lee, 2023. This training approach is called unsupervised learning. When training on unlabelled data from which ground truth can be inferred, the approach is called self-supervised data. The fine-tuning process on the other hand is mostly done on smaller datasets with supervised data, meaning they are labeled.

In the fine-tuning process, a pre-trained model, with its weights and biases already optimized, is trained again on additional data. This means that the model's weights will be slightly adjusted from their pre-trained state. The optimizer will find a new minimum in which the model will have learned additional information from the fine-tuning training data. When evaluated on test data, the newly fine-tuned model is expected to perform better than its pre-trained counterpart.

[9] The transformer architecture discussed in the previous section 5. But now it not only consists of an encoder but also has a decoder, which is responsible for the model's output.

6.3 Regression vs. seq2seq Models

Please note that the models fine-tuned in this section are not regression models but sequence-to-sequence models, which means that inside the model, tokens are processed, not numbers, like in the other models discussed. For the task in this project, specifically, fine-tuned models will output an integer rather than a float, because for seq2seq models solving an expression is a classification task, rather than a regression task. This is the reason why the only metric that makes sense for these models is accuracy, which is the percentage of correctly answered expressions from test data.

7 Findings

Lastly, the most noteworthy architectures were evaluated and scored on a benchmark. A custom benchmark was defined for this rather unique project, so that it takes into account the different generalization capabilities of models. For details, see the methodology, section: 2.3 The Benchmark.
An evaluation was performed on 5 training runs of each model, where the average of the 5 performances yielded the results in the following table[10]:

Regression models:	FNN2	FNN3	RNN2	Bidirectional LSTM	transformer4	transformer5
Total Parameters:	6'211	8'893	222'464	19'535	114'628	1'494'724
Architecture Parameters:	6'211	8'893	222,464	6'511	38'209	498'241
Optimizer Parameters:	-	-	-	13'024	76'419	996'483
MAE in Range:	0.026854	0.027486	0.244980	0.814113	0.039309	0.036588
MRE in Range:	0.010966	0.011122	0.093975	0.302951	0.016319	0.014936
MAE out Range:	2.178343	2.371700	3.737668	3.970515	4.419579	4.901888
MRE out Range:	0.250031	0.275299	0.363630	0.433860	0.495377	0.539891
MAE long Expressions:	6.155671	5.647490	5.519240	2.931804	3.886400	3.801507
Benchmark score:	10.702839	8.908331	0.665370	2.202380	5.023883	4.771119
Benchmark score p-value:	0.000	0.000	0.001	0.147	0.000	0.001

7.1 p-Value

The p-values[11] calculated symbolize the propability that the next benchmark calculated is ≤ 1 (or ≥ 1 if the model's benchmark is smaller than 1, which is the case for RNN2). Small p-values (< 0.05) signify that the models will perform better than the baseline model (or respectively worse in the case of RNN2). This proves the significance of the results.

To formalize this, let:

- $x_1, x_2, ..., x_n$ be the benchmark values where $n = 5$
- $y_i = \ln(x_i)$ be the log-transformed values for $i = 1, 2, ..., n$
- $\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i$ be the sample mean of the log-transformed values
- $s = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (y_i - \bar{y})^2}$ be the sample standard deviation
- $\mu_0 = 0$ be the hypothesized population mean
- t be the test statistic
- $P(condition)$ be the propability of a condition being true
- T_{n-1} follow a t-distribution with $n - 1$ degrees of freedom

[10] Mean Absolute Error (MAE) is the average deviation of the predicted answer, to the correct answer. And Mean Relative Error (MRE) is the deviation of the predicted answer, to the correct answer divided by the correct answer.
[11] Two-sided one-sample t-test was calculated on the natural logarithmic transformation of 5 benchmarks.

First described by Student, 1908, the p-value is calculated as:

$$t = \frac{\bar{y} - \mu_0}{s/\sqrt{n}} = \frac{\bar{y}}{s/\sqrt{n}}$$

$$p\text{-value} = 2 \cdot P(T_{n-1} \geq |t|)$$

7.2 Interpretaion

Firstly, notice how the FNNs dominate the benchmark, with the FNN2 being the best model out of the bunch. The p-values prove that all models except RNN2 and the Bidirectional LSTM with attention performed significantly better (with a 95% confidence interval) than the baseline model on the benchmark.
The bi-directional LSTM with attention is a very interesting model, because it performs bad on most datasets in comparison to other models, but it excels at longer expressions, which only slightly hinder the model's performance.
It is also notable that the main hypothesis – that more complex[12]models would perform better – is not supported by the results. This is made apparent by the clear outliers, such as the outstanding performance of the FNN2 (which is small in complexity) or the poor performance of the RNN2 model, which consists of the second to most parameters.
Also, notice the minimal improvement of the long expressions MAE between FNN2 and FNN3, which includes positional encoding. It can be derived that applying a positional encoding to the input only improves the model's performance when predicting longer expressions, on other test datasets used in this evaluation it only lead to a decrease in performance.

7.3 Fine-Tuned Models

The fine-tuned language models were also assessed based on their accuracy, here are the results from one training run each:

fine-tuned Language Models:	Gemini 2.5 Pro	Gemma 3 1B	Gemma 3 270M
Parameter size:	1.4E+11	1.00E+09	2.70E+08
Accuracy in Range:	95.17	93.41	54.22
Accuracy out Range:	99.51	63.33	9.67
Accuracy long expressions:	90	39.57	28

Notice how the less complex models with fewer parameters are less accurate. Additionally, only the performance of smaller models seems to be impacted by longer expressions and expressions with numbers outside of the range the models were fine-tuned on, Gemini 2.5 Pro even shows a better performance on the latter.

7.4 Guided Discussion Experiment

After conducting an open presentation for a colleague, his feedback was collected and his knowledge of the presented topics graded.
The starting hypothesis and the reason for conducting this discussion was to test if regression models (like the ones in this project) are easier to teach and understand for people interested in neural networks, who already possess some basic knowledge.
The Results from the questionnaire yield that the discussion partner did not fully understand either topic[13], but was able to answer basic questions. The topic of FNNs was easier to grasp than transformers.
According to the subject, the direct examples from this project which were delivered in accordance to theory, were the most helpful in forming an understanding.
The use of Regression models instead of sequence to sequence (seq2seq) models also helped. The reasons listed by the subject were the similarities between the regression in the output layer and linear regression as taught in school, as well as the usage of mostly numbers throughout the whole model architecture, from

[12]The complexity of a neural network depends in part on the number of parameters, and in part on the type of architecture, where a large number of parameters and a more advanced architecture correlate to a higher complexity
[13]The FNN and transformer architectures of regression models were taught.

input to output. The subject also stated that the latter helped with the understanding of tokens or vectors and how they are processed. Prior to this discussion this was not apparent to the subject.
The feedback included criticism about the lack of information and explanation regarding the optimization process. This is understandable, because the focus for this project was placed elsewhere.
for more details, see the questionnaire itself with the participant's responses and some comments by the author.
In conclusion, though only evaluated on a very small dataset, it can be said that it is beneficial to explain the basic workings of FNNs and transformers on the basis of models for regression, because of the wider use of numbers, as well as better connections to topics like linear regression, discussed as a part of the standard curriculum in school.

8 Discussion

8.1 Regression Models

All of the 4 architectures discussed here are very different, and all of them have different strengths and weaknesses, as one can tell by their performances on different sets of test data. Recurrent Neural Networks excel at long expressions, due to the way data is passed through these models sequentially in hidden-states. They are designed for handling longer inputs. Similarly, Bidirectional LSTMs with attention are even more effective when adjusting to a longer input. They can confidently predict longer expressions the best out of all the models studied in this project. This is because vanishing gradients are punished heavily when solving arithmetic expressions by nature, and this model combats this well, as opposed to the simplistic RNN architecture. Additionally, the attention mechanism not only helps with vanishing gradients in longer expressions but also emphasizes outstandingly large numbers and their corresponding signs.
When it comes to transformers, they perform best on data just like the one they are trained on, but not exactly the same; the classical definition of validation data. There are too many different types of different parameters, which all have been trained to process training data; this means even the smallest differences in input will lead to the model processing them wrong. Transformers are bad at generalizing rules from data, they perform the worst out of all models on expressions with numbers outside of the training range. Their decent performance on loger expressions shows that the attention is distributed well to numbers inside of the training range. transformer5 with more parameters than transformer4 doesn't show a performance improvement on the benchmark. This underlines that a more complex model doesn't necessarily mean a better performance for this task, rather the loss landscape is shaped with sweet-spots at the hyper parameters of transformer4 and transoformer5. Imagine it looking similar to the loss landscape drawn in figure 1.
The FNN architecture performed with the best benchmark. The FNN2 model was the best at expressions of the same length with numbers not encountered in the training data – The least complex model was best at generalization. Still, its abilities will not suffice to label it as 'able to generalize', primarily also because of its weakness at longer expressions. Even after adding positional encoding in FNN3, the model struggles at grasping expressions of different sizes.
A model's parameter count does not directly correlate with its benchmark performance. Instead, optimal performance is often achieved within specific parameter ranges. These "sweet-spots" can be identified through systematic methods such as hyperparameter optimization (i.e. with Keras-Tuner) or by empirically analyzing the performance of models of varying sizes (i.e. with a heatmap). In some cases, multiple performance optima may exist, as was observed with the transformer4 and transformer5 models.

8.2 Pre-trained Fine-tuned Transformers

The models in the lower table of the two are all very large, with Gemini 2.5 being particularly massive. Because we know that these models have the same transformer architecture and the sweet-spot for hyper-parameters is much smaller than their parameter size, we expect a weaker performance.[14]. Additionally,

[14]Since these are seq2seq models, their hyper-parameters sweet-spot would not be the same as the regression transformers. And their Decoder multiplies the number of parameters roughly by 2x. This is still not comparable to the jump in complexity between the largest optimized transformer regression model and the smallest pre-trained model, which is a factor of roughly 180x.

they predict on tokens rather than numerical values, which is not optimal for the task in this project. The performance of these models also depends heavily on the pre-training they previously underwent because it also involves arithmetics Wei et al., 2022. This explains why the bigger pre-trained models perform better – due to better pre-training. The Gemini 2.5 Pro model performed the best out of all models evaluated in this project on generalization tests, but this is most likely due to previous training, including expressions similar to those in the test data. For this reason it cannot be said that it is able to generalize.

8.3 Drop-Out

When introducing a Dropout with industry-standard values, contrary to the expectation of reducing overfitting (which is present to some extent, according to literature discussed in the literature review), the models are still not able to generalize beyond the training range, this is different to overfitting because validation data is being predicted with a simillar loss as training data.
The MSEs of models with Dropout are higher than those of the previous models without Dropout. This is because dropout effectively decreases the computing capacity of a model during training (when predicting, this is no longer the case) by deactivating a percentage of randomly chosen neurons in each layer. This explains their weak performance and why the KerasTuner usually prioritizes models without dropout.

8.4 Conclusion

The benchmark results expose that for generalization, less complex models tend to perform better than more complex ones. In the results, both the FNN models outperformed the more sophisticated transformer architecture models. FNNs are, overall, the best at generalizing and perform best on the defined benchmark. The downside is, that they struggle with longer expressions[15]. They are the best architecture for solving simple arithmetic expressions with a reasonable[16]supply of computational resources. Adding positional encoding barely makes a difference for FNNs, as is shown in the difference between the FNN2 and FNN3 models, it only slightly improves the performance on longer expressions. For this type of data sequential processing, like it is done in RNNs is best. While the simple RNN architecture is a weak model, bidirectional LSTMs with attention are better. By solving the vanishing gradient problem, the model solves longer expressions the best.
Transformers on the other hand, excel at learning patterns from training data and achieve strong performance on validation datasets; however, they exhibit reduced generalization capability compared to other model architectures. Optimal deployment requires training on large, diverse datasets that adequately represent the distribution of data the model will encounter during inference.
If maximum performance with unlimited computational resources is the goal, a fine-tuned, cutting-edge pre-trained transformer model such as Gemini 2.5 Pro will yield the best results.

Throughout evaluation the benchmark proved essential, to assessing the generalization capabilities of different models. This is thanks to the shifts in numbers used in the expressions, as well as the length of the expressions.

From the findings a broader implication for neural networks regardless of size or architecture can be drawn (though admittedly a little far-fetched):
Neural networks excel at interpolation within a the training distribution and struggle at extrapolating symbolic rules. This means neural networks should be treated as pattern-retrieval algorithms not rule-learners like our human brains.

Despite the chance factor playing a role, the conclusions will hold true. Conclusions were reached with significant margins and are backed by logical explanations. All results are reproducible with publicly available notebooks.

[15]and in general with longer, sequential inputs as shown by Goodfellow et al., 2016.
[16]Publicly available, affordable hardware e.g., The Nvidia Jetson Orin Nano Super Developer Kit GPU used in this project.

8.5 Possible Future Work

The topic chosen for this project is very broad and current. There's a lot of possible future work that can be done.

A logical and important continuation of this project is using the benchmark as the loss function; this might lead to the training of better-performing models, as well as a struggle with overfitting.

The fine-tuned models have been evaluated very sparsly in this study. A possible continuation would be to evaluate them in more detail. (In this study this wasn't done, because of the computational costs.)

The relatively small amount of data samples in the training data could why the neural networks were not able to generalize rules. One possible direction would be to use the same number range, but similar to how Trask et al., 2018 did it, use float-numbers in the training data instead of limiting yourself to only integers. This approax would allow for more training data.

Another possible direction is improving models by evaluating different activation functions, optimizers, or training data and using the best one, or even designing one's own. Training data and its tokenizer and padding is an area with lots of potential for improvement, i.e., by using an additional embedding, or finding the best training data, to teach a neural network simple arithmetics.

Also, consider this a proposal to evaluate a model architecture: an FNN with attention seems to be promising. Judging by the results of this study, this model is expected to improve the FNN's performance with longer expressions than in training, as well as strengthen its ability to accurately and confidently predict test data inside of the number range, previously referred to as the classical validation data.

A minimal but still interesting branch can be formed from this project to find the error or investigate the exact reason for the accuracy missmatch observed during and after fine-tuning.

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.
It was a very interesting project and a pleasure to work on. If you also 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, as well as Frau Smits for her availabilty to help.
Additionally a massive Thanks to my mom for reviewing the documentation on grammatical and logical inaccuracies.

References

Amidi, A., & Amidi, S. (2019). Recurrent neural networks cheatsheet [Online; accessed November 11, 2025]. https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

Bahdanau, D., Cho, K., & Bengio, Y. (2016). Neural machine translation by jointly learning to align and translate. https://arxiv.org/abs/1409.0473

Bergmann, D. (n.d.). What is fine-tuning? https://www.ibm.com/think/topics/fine-tuning

Cho, K., van Merrienboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. https://arxiv.org/abs/1409.1259

Cristina, S. (2023, January). The bahdanau attention mechanism [Accessed: 2025-10-11]. https://machinelearningmastery.com/the-bahdanau-attention-mechanism

Emil. (2020, February). What is the feedforward network in a transformer trained on? [Accessed: 2025-11-03]. https://datascience.stackexchange.com/questions/68020/what-is-the-feedforward-network-in-a-transformer-trained-on

Gawde, R. (2021). Image caption generation methodologies. https://www.researchgate.net/figure/Fig-3-RNN-A-recurrent-neural-network-RNN-is-a-class-of-artificial-neural-networks_fig1_351840108

GeeksforGeeks. (2024, August). Large language models (llms) vs transformers - geeksforgeeks. https://www.geeksforgeeks.org/nlp/large-language-models-llms-vs-transformers

GeeksforGeeks. (2025a, April). Deep learning introduction to long short term memory [Last Updated: 2025-04-05]. https://www.geeksforgeeks.org/deep-learning/deep-learning-introduction-to-long-short-term-memory

GeeksforGeeks. (2025b, October). Gated recurrent unit networks [Last Updated: 2025-10-09]. https://www.geeksforgeeks.org/machine-learning/gated-recurrent-unit-networks

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning* [http://www.deeplearningbook.org]. MIT Press.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Kim, S., Nam, H., Kim, J., & Jung, K. (2021). Neural sequence-to-grid module for learning symbolic rules. https://arxiv.org/abs/2101.04921

Lee, K. (2023). Open-sourced training datasets for large language models (llms) [Accessed on 2025-11-03]. https://kili-technology.com/large-language-models-llms/9-open-sourced-datasets-for-training-large-language-models

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

Pascanu, R., Mikolov, T., & Bengio, Y. (2013). On the difficulty of training recurrent neural networks. https://arxiv.org/abs/1211.5063

Srinivasan Anusha, J. (2024, December). Transformer architecture — the backbone of llms [Accessed: 2025-10-12]. https://jananithinks.medium.com/transformer-architecture-the-backbone-of-llms-1a3d085ca981

Stryker, C. (n.d.). What are large language models (llms)? https://www.ibm.com/think/topics/large-language-models

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

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

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E. H., Hashimoto, T., Vinyals, O., Liang, P., Dean, J., & Fedus, W. (2022). Emergent abilities of large language models. https://arxiv.org/abs/2206.07682

Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., ... Wen, J.-R. (2025). A survey of large language models. https://arxiv.org/abs/2303.18223