Neural Predictive Calculator

Findings

Maturitätsarbeit, Kantonsschule Baden
Michael Schneider, Julia Smits

Anton Mukin G4h

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. Still, no model performed well enough to be classified as *having learned* arithmetic rules.

The regression models were presented to a colleague in a guided discussion. Though only on a small and insignificant dataset, it was found to be beneficial to explain the basic workings of FNNs and transformers.

0.1 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.

0.2 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 embeddings, a seq2grid pre-processor[1] and a PReLU activation function were expected to help the model generalize simple arithmetic rules.

[1]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.

List of Figures

Contents

1 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 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.

2.1 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)

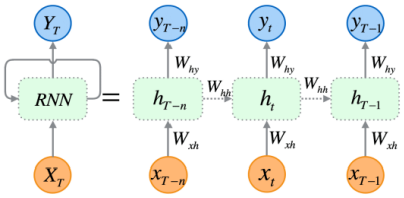

Figure 1: RNN diagram as described by Gawde, 2021

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

2.2 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.

3 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.

3.1 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.

3.2 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.

3.3 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), it is then normalized (13). Finally the attention weights scale their respective hidden-states from the LSTMs, to compute the context vector (14).

4 Transformers

The best performing and most widely used architectures for most tasks to date are transformers; 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.

4.1 Multi-Head Self-Attention

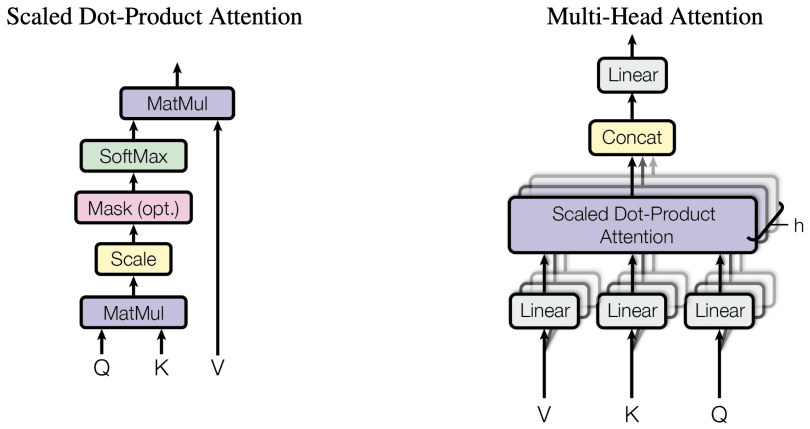

Figure 2: Two diagrams from Vaswani et al., 2023 with the scaled dot-product attention described on the left and multi-head attention on the right.

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[2] 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.[3] (15)

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

$$\text{where}^{[4]}\text{: } \text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K$$

This is done across all heads in parallel. The results are then concatenated[5] back together and they undergo a linear projection. As described below:

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

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

[3] K has a T, 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.

[4] 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.

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

4.2 The Encoder Layer

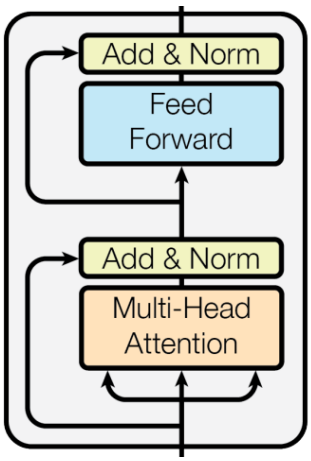

Figure 3: Encoder Layer as described by Vaswani et al., 2023

In figure 3 the multi-head attention first undergoes a residual connection as well as a layer normalization[6], 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.

4.3 Point-wise Feed-Forward Network

Equation (17) and figure 4: 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 3. 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 3.
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.

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

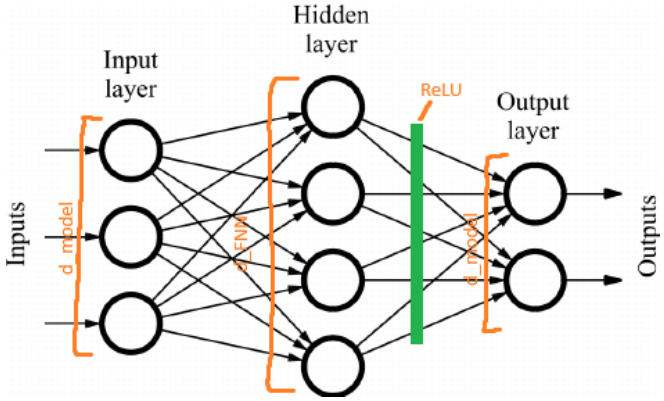

Figure 4: A diagram from Emil, 2020 representing the structure of a pointwise FNN.

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

4.4 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.

5 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.

5.1 LLM

LLMs adopt the transformer architecture[7] 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.

5.2 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.

[7] The transformer architecture discussed in the previous section 4. But now it not only consists of an encoder but also has a decoder, which is responsible for the model's output

5.3 In Addition

Please note that these are not regression models but sequence-to-sequence models, which means that inside the model, tokens are processed, not numbers, as in the other models discussed. For the task in this project, specifically, fine-tuned models will output an integer rather than a float, because tokens for floats have not been created. This is the reason why the only metric that makes sense for these models is accuracy, which essentially evaluates how many times the model predicted correctly.

6 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: 1.5 The Benchmark.
The results[8]were documented in this table:

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:	0.0389406	0.034600	0.2730647	0.5908327	0.0448075	0.0381983
MRE in Range:	0.0155504	0.013438	0.1135200	0.2712299	0.0188047	0.0151836
MAE out Range:	2.2301364	2.504607	3.5727410	3.2350100	4.5702314	4.9287004
MRE out Range:	0.2509270	0.290885	0.3472042	0.3404062	0.5201459	0.5431157
MAE long Expressions:	6.2929916	6.116872	6.0766930	2.7806435	6.1630590	6.0680327
Benchmark score:	6.6345832	6.672521	0.4793787	1.7374969	2.8882827	5.2896368

And the fine-tuned Language Models, evaluated on their accuracy:

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

Firstly, it is notable that the main hypothesis – that more complex[9]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 or the poor performance of the RNN2 model. Also, notice the minimal improvement between FNN2 and FNN3, which includes a positional embedding layer. It is visible that the FNN models show the best performance on the benchmark, meaning they are crowned the winners.

6.1 Guided Discussion

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 posses some basic knowledge.
The Results from the questionnaire yield that the discussion partner did not fully understand either topic[10], 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 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.
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]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.
[9]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
[10]The FNN and transformer architectures of regression models were taught.

7 Discussion

7.1 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. transformer5 with more parameters than transformer4 only shows a significant performance improvement on the classical validation data. Thanks to positional encoding, transformers can handle longer inputs better than FNNs.
The FNN architecture performed with the best benchmark. The FNN2 model was good at expressions of the same length with numbers not encountered in the training data. 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 embedding in FNN3, the model struggles at grasping expressions of different sizes.
In addition it can also be said, that the number of parameters in a specific model architecture does not directly correlate to its benchmark performance, rather there are sweetspots for the number of parameters that can be found using the Keras-Tuner or a heatmap of differently sized models and their performances. It can be the case sometimes that there are multiple sweetspots, like it was the case for transformer4 and transformer5, who both were trained with Keras-Tuner.

7.2 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.[11]. Additionally, 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. This explains why the bigger pre-trained models perform better. The Gemini 2.5 Pro model performed the best out of all models evaluated in this project on generalization tests, but this is due to previous training, including expressions similar to those in the training data. For this reason it cannot be said that it is able to generalize.

7.3 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 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,

[11] 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.

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 prioritizes models without dropout.

7.4 Conclusion

Adding positional embedding barely makes a difference for FNNs. RNNs are weak models, outclassed by bidirectional LSTMs with attention, and should not be used. Transformers excel at accurately learning the training data but struggle with generalization more than other models. FNNs are, overall, the best at generalizing and perform best on the defined benchmark. They are the best architecture for solving simple arithmetic expressions with a reasonable[12]supply of computational resources. If maximum performance with unlimited computational resources is the goal, a fine-tuned, cutting-edge pre-trained model such as Gemini 2.5 Pro will yield the best results.
Despite the chance factor playing a role, the conclusion will hold true. Conclusions were reached with significant margins and are backed by logical explanations. All results are reproducible with publicly available notebooks.

7.5 Possible Future Work

The topic chosen for this project is very grand and current. 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.
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.

[12]Publicly available, affordable hardware e.g., The Nvidia Jetson Orin Nano Super Developer Kit GPU used in this project.

References

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.

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

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

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

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