Methodology for Predictive calculator

Anton Mukin

June 2025

0.1 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. 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 to be the performance of a basic FNN's performance on different, but roughly still similar arithmetic tasks.
Afterwards, the same was done for the transformer type of neural network model. Here, their exact functionality was thuroughly investigated, because of their unique architectures.
Lastly a simillar workflow was repeated for some bigger, pre-trained models.
And finally all the findings were collected and evaluated as a whole.

0.2 Introduction to this document

The goal of this document is assisting reproducability and showing how the findings discussed in the other document have been obtained.
All of the code written for this project is available in the github reporsitory:
https://github.com/AntonStantan/matura
In this project all of the code is written in Python-notebooks (Jupyter Lab).
The preffered library used was tensorflow keras.
Most of the models were trained locally on a Nvidia GPU device: *Nvidia Jetson Orin Nano Super Developer Kit*

Figure 1: You can see the aformentioned Nvidia Jetson device booting up.

Contents

1 Feed-forward Neural Networks (FNN)

For details on how a FNN works please refer to the section two in the literature study.
This section refers to the /FNN directory in the github reporsitory.
There are two notebooks here: FNN1 and FNN2.

1.1 Train and Test data

Defining the train data; The decision was made to only use subtraction and addition. The arithmetic expressions were defined to consist of two operators (+ or -) and three integers $[-5, 5]$
The reason for these definitions are: + and - are the 2 simplest operators. And the decision to introduce the model to 3 numbers, in place of the more commonly used 2, was made in hopes of simplifying the transition to more numbers later on.
e.g., the first three entries are:

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

In total there are 1907 expressions like this in the train data.

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

- Inside of the number range: Expressions just like in the train data but not inside of train data.
- Outside of the number range: The same expressions, 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: Expressions of different lengths. Specifically, there are between 2 and 8 numbers inside of the $[-5, 5]$ range. e.g.,

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

1.2 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 start of we will import the necessary libraries.

```
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
```

And then we have to make a dataset out of the train and validation data discussed in the previous subsection. Let's use batches of 32. Validation data is like the In-Range-test-data, except there aren't as many 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)
```

We can now define the model 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 is the one, which was the most promising in Trask et al., 2018.
Drop-out is included to prevent overfitting. In most models used in this project it wasn't necessary.
As you can see, the input-shape has to be defined, when defining the model.
Additionally, please notice how the output layer only consists of a single neuron with a linear activation function. This means the model we just created is a regression model. This 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 loss calculation e.g., in our case Mean Squared Error (MSE)

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

The last step remaining is to fit the model on some data. We will be fitting our model on the previously defined train and validation data. The training process will take 200 epochs to complete or it might be aborted preemtively 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
)
```

1.3 FNN1 and FNN2 Notebooks

In this project two notebooks with FNNs have been created. FNN1 and FNN2. In FNN1 a simple FNN was created, it's performance was evaluated and FNNs of different sizes have been compared against eachother in a heatmap. It is noteworthy, that the models here have been 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 keras-tuner. This is simply put an automatization: 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 of that notebook and it's benchmark is calculated.

1.4 The Benchmark

To calculate the benchmark a model is evaluated on 4 categories: The test data inside of the number range, outside of the number range, longer expressions and a relative MSE of the test data with numbers outside of the range. Refer to the code below:

```
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 usage of a relative error punishes mistakes on "simpler" expressions and is less harsh on more "difficult". 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: Not all expressions outside the number range, and not all longer expressions were used.
For longer expressions only expressions with 4 numbers were used.
For expressions with numbers outside of 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, aswell as the inlusion of the relative MSE is done in hopes of bringing the benchmarks of different models closer together and to increase the linearity between them. This makes the benchmark more comfortable to work with.
The baseline values are defined to be of a FNN model with 30 neurons and two dense layers. It is trained over the period of 200 epochs with early_stopping enabled.
As clearly observable in the code for calculating the benchmark, using this formula defines a model with the same performance as the baseline model to have a benchmark of 1. Models performing worse or better on the test data yield scores below or above 1, respectively.

1.5 Drop-Out

When introducing a Drop-Out with an industry standard value of 0.3, contrary to the expectation of reducing over-fitting, which in some way is present (according to literature discussed in the literature study), as the models aren't able to generalize beyond the training range, this has a negative effect on the model. The MSEs of models with Drop-out are higher then, the ones of the previous models without drop-out.
This is because drop out effectively decreases the computing cpacity of a model during training (when predicting this is no longer the case), by deactivating a percentage of randomly chosen neurons in each layer.

2 Recurrent Neural Network (RNN)

Recurrent Neural Networks (RNNs) work similar to FNNs with one key difference: There is a vector called the hidden-state. This vector contains information about previous inputs. 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 hiddenstate 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 (commonly tanh or ReLU for the hidden state)
- σ_{out}: Activation function for the output (e.g. softmax for classification, or linear in our case of regression)

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = \sigma_{out}(W_{hy}h_t + b_y)$$

2.2 RNN0 and RNN2

The RNN0 notebook includes a model built with the aformentioned architecture. A 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")
])
```

Notice 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 like the stock market.
The same notebook used in FNN2 was adapted to work with RNNs in the notebook RNN2. Like in FNN2 the keras-tuner was used for finding the optimal number of neurons, as well as whether or not to include the dropout after a dense layer.
Ensuring the keras-tuner can choose to use a dropout isn't crucial, because we expect all results from a model with a drop-out to be worse. This only helps if the model would be over-fitting otherwise.
Useful sources for the creation of the first RNN prototype: Bowman et al., 2015; "What is a recurrent neural network (RNN?)", n.d.; Zhu and Chollet, 2023

3 Attention and Transformers

3.1 Attentional RNNs

For the sake of transitioning from RNNs to transformers an attentional RNN model was trained and evaluated.
It consisted of a bidirectional[1] 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.
A number of 35 LSTM units were chosen for the Encoder because of it's previous performance (with bootstrapping), visible in a heatmap.

[1] 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 context representation

3.2 Transformers

For the example of which architecture to use for building a transformer, we turn to the original paper Vaswani et al., 2023. The architecture used in this project resembles the one that first introduced the transformer architecture with one small difference: It is a Encoder-only model, not the sequence to sequence type from the paper.

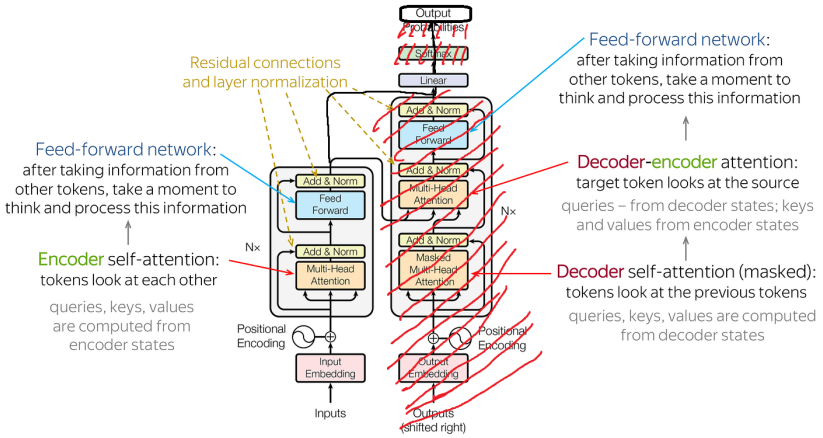

Figure 2: A Seq2Seq model from Vaswani et al., 2023. The Encoder-only model used in this project, is the same, except the decoder part is skipped, in the image this part has been crossed out with red.

To code this in python an object oriented approach has been taken, with classes consisting of other classes, like a matryoshka:

- Multi-Head Attention (MHA) and pointwise-FNN
- Encoding layer consisting of the MHA and pointwise-FNN, as well as layer normalization and dropout (included in the original paper to prevent drop-out)
- Encoder containing encoding layers and drop-out.
- Transformer which encapsulates embedding and positional encoding, as well as obviously the encoder and a final one-neuron-FNN with a linear activation.

A model with the same hyperparameters like the ones used by Vaswani et al., 2023 can be found in transformer0.
Sadly this model get's stuck in a local minimum when training, as you can tell at the bottom of it's notebook. The minimum is just predicting a number close to 0 for every expression.
We don't want this, so we have to tune the model's hyperparameters. We do this with the help of keras-tuner.

A couple of other minor adjustments were made to the model, mainly switching to AdamW optimizer, because it is better than Adam for transformers Loshchilov and Hutter, 2019. And using a learning rate scheduler with warmup and Cosine decay. This means that not like before, when the learning rate stayed constant throughout the training, the learningrate changes. It is higher in the beginning and decays towards the end. This helps the model take "big steps", "jump over" local minimums and take more mindful, careful "steps" towards the end of the training.

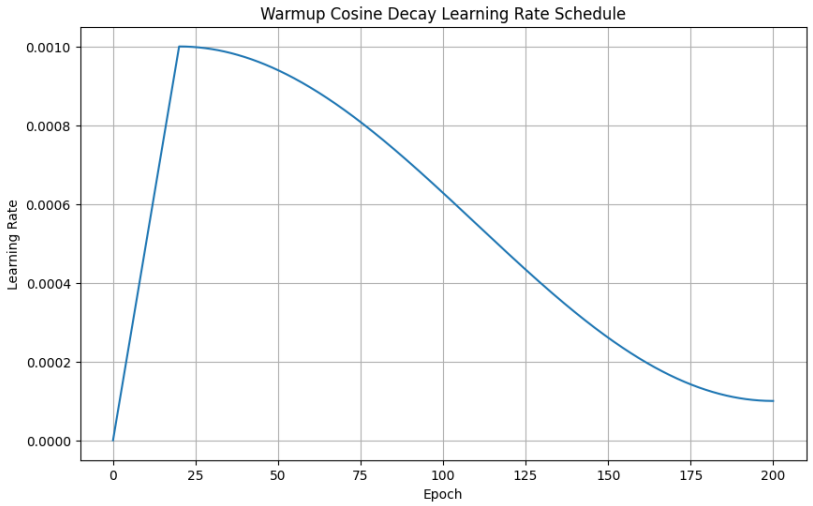

Figure 3: This is how the learning rate could look like for a model being trained over the period of 200 epochs, when using a cosine decay with a linear warmup, like in this project. (Except for the changing peak-learningrate all other parameters are the same as the ones used in this project.)

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 tensors have inside the model
- The number of encoding layers
- The number of neurons in the layer inside of the pointwise-FNN
- Wether 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 Transformer5 is a bigger model, than Transformer4.

4 Pre-trained Transformers

The only model-type left for me to analyse were fine-tuned pre-trained LLMs. Since working with them is completely different than working with your own locally trained neural networks, a lot of research had to be done before beginning to dig into the work.

A lot of fuss went into library compatibility for the amd64 architecture system on the Jetson device. The final solution was using a the jetson-containers who 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`

4.1 Gemini 2.5 Pro

The first model to be fine-tuned was Gemini 2.5 pro. It is the one I personally use the most, it was also chosen in part because of it's 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 from the VertexAI website. First a .json file with the training data had to be created and uploaded. Afterwards the model was fine-tuned in the notebook gemini_vertex. 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 pretty simplistic notebook is created: gemini_vertex_predict
There, the model is loaded in from it's endpoint and used to predict test data.

4.2 Gemma 3

For some diversity a smaller (Gemma3 1B parameters) and a tiny (Gemma3 270M parameters) were chosen.
Because of their size, they can be fine-tuned locally on the jetson-device. They are fine-tuned with the SFTTrainer provided by trl.[2] The guide on huggingface was used as help for this.
During the fine-tuning process, a custom designed function that calculates the accuracy on some validation data (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:

[2]We cannot use the standard Trainer from huggingface, because the finetuning we do is supervised, hence the name: Supervised Fine-Tuning (SFT)

Gemma3 1B, Gemma3 270M

After fine-tuning the models can be found in the hugggingface repositories:
fine-tuned Gemma3 1B, fine-tuned Gemma3 270M
After a basemodel 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.

4.3 Weird Results Encountered After Evaluation

A weird observation rose up: The accuracy during the fine-tuning process is greater than the accuracy calculated when evaluating the models in the prediction notebooks. This is probably due to some error that was overlooked but no errors were found.
The error was approached from a systematic standpoint; Multiple hypothesis have been set and all of them disproven. The error (if there is one) could not be found.

Hypothesis for reason of the error:

- There is a previous model that is being loaded, instead of the most recent one. (or not the most recent epoch)

All the previous runs have been deleted, there is only one left. The specific checkpoint has been specified (the most recent one).

- There is an error with generating the data, like using different hyperparameters (temperature and stuff.)

The pytorch .generate() and the huggingface provided generator pipeline both yielded simmilar results. And the pipeline uses hyperparameters from the fine-tune.

- There might be an issue with the calculation during the evaluation steps

I thuroughly checked the compute_metrics function, and made sure the validation dataset is being used and not the train dataset. The function itself is very basic and there is no error, as confirmed by multiple LLMs.

- The model is overfitted

No, predicting on train data yielded the same (slightly better) results

After trying out different checkpoints, all of a sudden the accuracy of the 270m model dropped from 40

- This lead me to believe maybe it has something to do with the way the models are calculating everything, apart from the temperature and other hyperparameters, it might be because of performance optimization software calculation on GPU or GPU with work offload to CPU might give different results.

Transformers is imported the same way, and is working on GPU exclusively both times.

- Ran the code on the exact same dataset used in the validation part during training. no differences.

- There is a slight difference with the calculation. In the compute metrics function the prediction is only counted if the model uses a turnseparator. This could explain why the seemingly normal gap in accuracies occurred.

> After rerunning the training with the correction, the accuracy dropped a little (from 100
>
> - The pre-trained model obviously has a drop out and layer norms. The issue might be that the model is not in eval mode and is dropping out a lot of valuable data, this would explain the slightly worse performance.
>
> This was not the case, because after setting model.eval() this only increased the accuracy by around half a percent for the Gemma3 270M model, nothing really changed. + This should probably be on by default anyways.
>
> - A perfectly reasonable explanation would be, the model's weights aren't loaded in to the model properly, like it happened with FNN2.ipynb *see 4th of Oct Arbeitsprozess.
>
> This cannot be the case because loading in the hyperparameters is completely different to loading in the weights. A model with loaded-in weights is already trained (hence the name - pre-trained model). Additionally the model is showing signs of having learned from the fine-tuning process for example in the way it responds with the trained Int.0 instead of the traditional Int.

In case you find out why this happens, please contact the author of this project.

5 Closing Remark

The author put a lot of work into this project, he would appreciate all sorts of feedback anywhere: from email: anton.mukin@students.ksba.ch, to discord: stantan2
Please leave a star on the github repo if you found it interesting.

I want to thank Herr Schneider for the great insights and quick 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

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

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

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

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