Findings

Anton Mukin

October 12, 2025

0.1 Abstract

0.2 Introduction

This project started with a simple Idea – A calculator that rather than working systematically, predicts your answer using a neural network. The question which neural network architecture would fit this job best is what what truly propelled this project.

This document presents the findings, which have been learned with the project: A Predictive Calculator.

0.3 Feed-forward Neural Networks (FNNs)

Since the functionality of a FNN has already been discussed in the literature study, were mentioned in the methodology document and no new information on FNNs has been found since then by the author, this topic will not be discussed in this document.

Though, FNNs will make an appearance later on in this document.

Contents

1 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.

1.1 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 regression)

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

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

1.2 Relevant Takeaway

For us this means, the RNN doesn't process an expression as a whole, but rather one part after another. Also, because of how the formula works, Tokens, which are later in the sequence, play a more impactful role on the models 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, not only prevelant in this project. It is more widely known as the vanishing gradient problem.

2 Other Types of RNNs

To solve the problem described above, different methods have been adopted like 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 even the concept of attention proposed by Bahdanau et al., 2016.

2.1 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 you 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.

2.2 Gated Recurrent Unit (GRU)

The GRU architecture works in a similar way to the LSTM. Instead of utilizing memory cells, they directly use the hiddenstate.
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 step

4

- h_t is the current hiddenstep

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

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 step to calculate the candidate hidden state (9).
Lastly the hidden step can be calculated by applying (1 - the update gate activation) to the previous hidden step 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 step or the candidate hiddenstate will weigh in more on the current hidden step.

2.3   Bidirectional LSTM with Attention

An architecture of this type consists in part of a bidirectional LSTM, meaning two LSTMs working in parallel, one of who process data from front to back, the other from back to front, their results are afterwards concatenated together 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 hiddenstates for each timestep t, the output from the bidirectional LSTM,

- $\mathbf{W} \in \mathbb{R}^{128 \times 128}$ 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 hiddenstates from the LSTMs, to compute the context vector (14).

3   Transformers

The best performing and widely used on most tasks to date are transformer architectures, they formed the basis for LLM's.
The key to their success: multi-head self-attention.

In this document we will discuss the transformer architecture as first proposed by Vaswani et al., 2023.
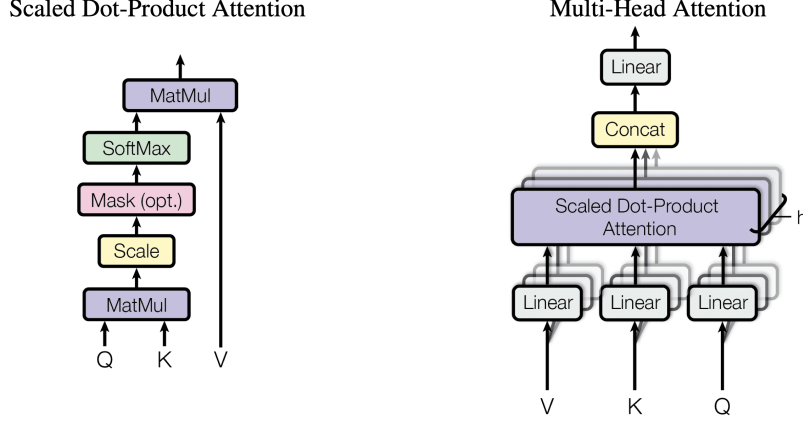
3.1   Multi-Head Self-Attention

Figure 1: 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 up eqally into multiple heads which will process it in parallel. There, the tensors are linearly projected with trainable weights, to obtain queries (Q), keys (K) and values (V) which you can see 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[1] to determine how simmilar they are. This leaves us with a number between 0 and 1 representing how much attention you have to pay. The value V represents the actual information of the token we just calculated the attention for. This means our final step is to scale the value V by it's attention. Pay attention to the equation below. (15)

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

This is done across all heads in parallel. The results are then concatenated[2] back together and they undergo a linear projection. As described below:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

$$\text{where: } \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

---

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

[2]This means the are assembled or added back together, to have the same dimensions as before they were split up.
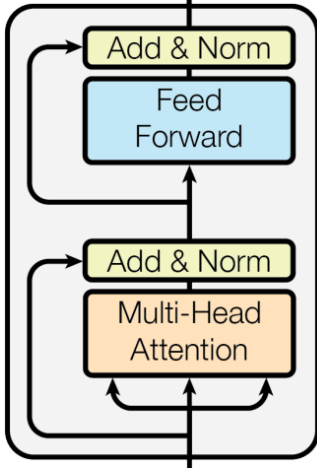
3.2 The Encoder Layer

Figure 2: Encoder Layer as described by Vaswani et al., 2023

In figure 2 the multi-head attention first undergoes a residual connection as well as a layer normalization [3], as described below:

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

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

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.

3.3 Point-wise Feed-Forward Network

Equation (17) and figure 3:
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 vlaues and the output layer with weights and biases W_2 and b_1 resets the data back to it's original dimensionality.

After undergoing a residual connection and layer normalization like after multihead attention (16), this makes up a single layer of the Encoder.

A regressive transformer model, like the one used in this project, consists of multiple such encoding layers and last but not least a FNN with a single neuron, which works as the output layer.

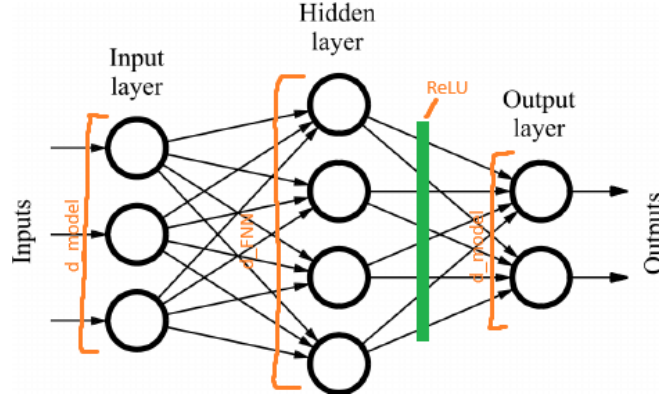$$\text{pointwiseFNN}(x) = \max\big(0,\ xW_1 + b_1\big)W_2 + b_2 \tag{17}$$

Figure 3: A diagram representing the structure of a pointwise FNN. source for figure

3.4 Positional Encoding

Something, that's special about the tranformer, is the fact that data is passed through it as a whole. This means the model doesn't have positional understanding on it's own. To counteract this effect, a positional encoding is applied to the input sequences after the tokenizer.

---

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

References

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

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

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

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