Neural Network with Back Propagation for Symbolic Mathematics

Mary Grace MacDonald

University of Waterloo

MTE 203

November 24th, 2023

I. Introduction

Symbolic mathematics can express complex mathematical relationships using symbols, enabling abstraction and generalization of solutions. Unlike numerical methods, symbolic systems provide exact and general solutions, making them invaluable in solving equations, manipulating algebraic expressions, and handling advanced mathematical concepts. The power of symbolic mathematics lies in its capacity to represent and explore theoretical mathematical structures, aiding in a deeper understanding of mathematical principles. This approach, widely used in scientific research and engineering applications, allows for precise modelling, simulation, and analysis of complex systems. Additionally, symbolic mathematics is an educational tool, facilitating a more intuitive grasp of abstract mathematical concepts and promoting critical thinking skills among students.

As a language of precision and abstraction, mathematics often involves intricate algebraic expressions, differential equations, and symbolic manipulations. By utilizing neural networks, machines can process and interpret the intricacies of symbolic mathematics. A study applying deep learning to symbolic mathematics [1] shows that solving symbolic integration and differential equations with deep learning may be faster and more accurate than currently existing symbolic manipulators such as Wolfram Mathematica, MATLAB, and Maple.

A. Objective

This report aims to develop and describe a neural network capable of demonstrating a learned understanding of symbolic integration.

II. Background

Symbolic mathematics, also known as symbolic computation or algebraic computation, is a branch of mathematics that deals with manipulating mathematical expressions and equations in symbolic form [2].

A neural network is a computational model inspired by the intricate structure of the human brain. Like the brain processes information through interconnected neurons, neural networks consist of layers of artificial neurons that work together to learn and make predictions from data [3].

Natural Language Processing (NLP) is a set of computation techniques that analyze language. In this report, a language is a finite set of symbols or tokens, also known as vocabulary. NLP requires high-level symbolic capabilities to extract meaningful information from language structures [4]. Using NLP, neural networks can process the complexities of written languages – this also applies to symbolic mathematics [1].

A. The Model

The transformer architecture is a way to implement a sequence-to-sequence (seq2seq) neural network using the attention mechanism, also known as the scaled dot-product attention [5]. The purpose of the transformer is to provide contextual awareness for each of the tokens in the sequence. Seq2seq is designed for tasks where the input and output are data sequences. In NLP the input and output can be considered a sequence of symbols.

Seq2seq consists of two main components: Input is provided to the encoder, which processes the input step-by-step and produces a fixed-size context vector; The decoder takes the context vector and generates the output sequence step-by-step. This architecture can be seen in Figure 1.

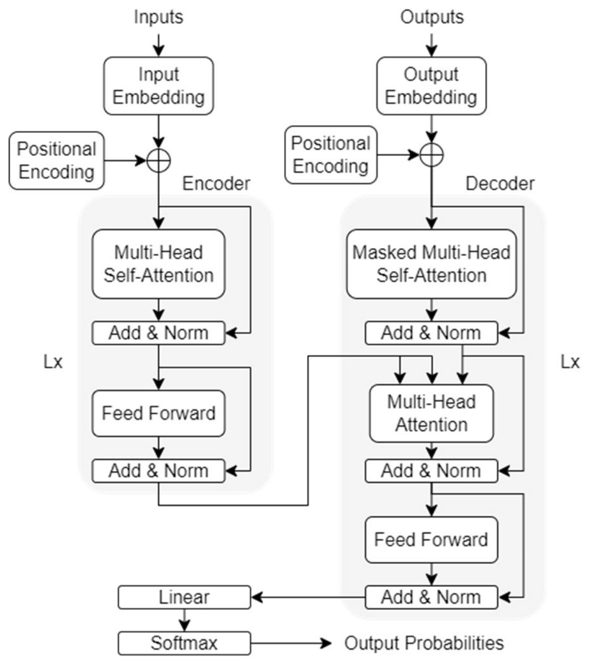


Figure 1. The Transformer architecture [6]

Linear Transformation

A perceptron is a type of artificial neuron that takes n inputs x, applies a weight w to each input, and sums them to produce a output

$$x' = \sum_{i=1}^{n} x_i w_i \qquad (1)$$

as seen in Figure 2.

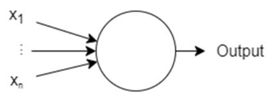


Figure 2. Visual representation of a perceptron [3]

A linear layer takes in an n-sized vector and performs linear transformation, also called linear mapping, to produce an m-sized vector. It is composed of m perceptrons as seen in Figure 3.

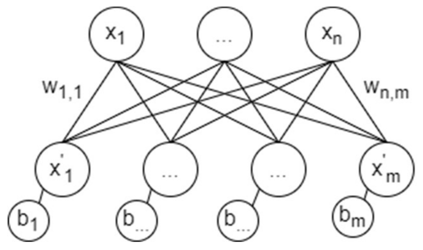


Figure 3. A linear layer with n inputs and m outputs

Each layer has a weighted matrix $W \in \mathbb{R}^{n\times m}$ that defines the linear transformation. The weights in W are learned by the network using backpropagation. Given an input vector $x \in \mathbb{R}^n$, a linear layer can be represented as

$$x' = x \times W + b \tag{2}$$

where $b \in \mathbb{R}^m$ is the bias vector, which allows the model to capture offsets or shifts in the data that are not solely determined by the input features [3].

Embeddings and Positional Encoding

Embeddings use a linear layer without biases to convert each of the input tokens to a d_{model}-dimensional vector that the neural network can interact with [5]. The weight matrix for the embedding W_E has $|V|$ rows and d_{model} columns, the vocabulary size, and the dimensionality of the model respectively [7]. Each row in the embedding matrix represents the embedding weights for a specific token. The encoder and decoder each have a respective embedding matrix.

The dimensionality of the transformer represents one dimension of the transformer's complexity. In general, transformers with a larger d_{model} allow the model to represent more intricate relationships within the input sequences. Increasing d_{model} also increases the computational complexity of running the transformer [6].

The input sequence is padded such that empty vectors beyond the last token in the sequence are added, up to the maximum sequence length N [5]. This allows for fixed length input, so linear algebra operations can function. Each token is given a distinct vector $t_i \in \mathbb{R}^{d_{\text{model}}}$ from the i^{th} row of the embedding matrix. The input tokens can be collected to form a matrix $T \in \mathbb{R}^{N\times d_{\text{model}}}$, where each token vector corresponds to a row in the matrix [6].

Semantical significance corresponds to how related each token is to other tokens. In other words,

$$\text{Similarity} = \frac{t_i \cdot t_j}{|t_i||t_j|} = \cos(\phi) \tag{3}$$

where ϕ is the angle between the two arbitrary embedding vectors t_i, t_j. Note that as the similarity approaches one, the tokens become similar and conversely as the similarity approaches negative one, the tokens become dissimilar [7].

A positional encoding matrix $P \in \mathbb{R}^{N\times d_{\text{model}}}$ is added to each embedding matrix to account for the order of the sequence. This gives the embedded matrix

$$T' = T + P \tag{4}$$

where each row of the matrix corresponds to an embedded token t'_i. The elements in the positional encoding matrix are calculated using

$$P_{[pos,2i]} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{5}$$

for even indices and

$$P_{[pos,2i+1]} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{6}$$

for odd indices, where pos is the position and i is the current dimension [5].

As seen in (5) and (6), the period of the sinusoidal function increases as the dimension of the vector i increases to d_{model}. This ensures that the positional encoding at each distinct position is unique and offsets the input vector to consider positional significance as well as semantical significance. Note that for any sequence, the positional embeddings will remain the same at each position [5].

Another way to represent the significance of each embedding vector is represent each token vector in $|V|$-dimensional space. Related token vectors, vectors with similar weights, will be closer together in the embedding space. The positional embedding is pushing each vector toward a space which clusters tokens based on their position [8], thus increasing their similarity. The embedding space is visualized in Figure 4.

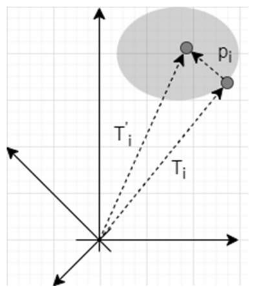


Figure 4. Example of a $|V|$-dimensional embedding space

Encoder and Decoder Stacks

In this architecture, as seen in Figure 1, there are two stacks of layers, the encoder, and the decoder. Both the encoder and the decoder consist of Lx identical stacks as seen in Figure 1. These stacks are run sequentially, where the output of one layer becomes the input of the consecutive layer.

The encoder is responsible for processing the input sequence and its representations. Each layer consists of two sub-layers; the first being a multi-head self-attention mechanism, and the second a position-wise feedforward network. Between each sub-layer a residual connection is applied followed by layer normalization [5].

The decoder layer generates the output sequence based on the representations learned by the encoder. In addition to the two sub-layers in the encoder, the decoder has a third sub-layer which performs multi-head attention over the output of the encoder stack. A transformer is autoregressive, meaning each generated token is dependent on its own past values. Additionally, the self-attention mechanism in the decoder is designed such that for each generated output token, the mechanism prevents context from subsequent tokens which have not yet been generated [5].

Layer Normalization

Layer normalization is a type of normalization technique used in transformers and other neural network architectures. It is applied at the level of individual layers, normalizing the inputs for each token within a layer independently.

$$\text{LayerNorm}(x) = \frac{x - \bar{x}}{\sqrt{\sigma}} \tag{7}$$

where x is the input tensor, $\bar{x}$ is the mean of the tensor, and σ is the variance. The purpose of layer normalization is to address issues related to internal covariate shift, which can occur during training when the distribution of inputs to a layer changes [3].

Self Dot-Product Attention

The attention mechanism allows the model to selectively focus on different parts of the input sequence when producing each element of the output sequence. Each token is associated with a query vector, and a set of key-value pairs as vectors. An output vector is computed as the weighted sum of the values, where the weight assigned to each value is computed from the query and key vectors [5].

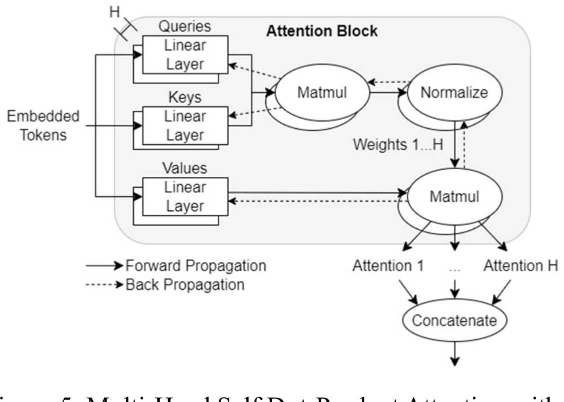


Figure 5. Multi-Head Self Dot-Product Attention with H heads

The model uses multi-head scaled dot-product attention as shown in Figure 5. In scaled dot-product attention, the embedded matrix X is mapped to a query, key, and value matrix, $Q, K, V \in \mathbb{R}^{N \times d_{\text{model}}}$ respectively [5].

The mapping is the result the linear transformation

$$(Q)_{i,j} = \sum_{k=1}^{n} (T')_{i,j} \left(W_Q\right)_{k,j} = T' W_Q \tag{8}$$

where $W_Q \in \mathbb{R}^{N \times d_{model}}$ corresponds to a learned weight matrix for the queries [9]. (8) is performed for the query, key, and values with respective weight matrices. The mapped matrices are used to compute the context matrix

$$\text{Attention}(\text{Q}, \text{K}, \text{V}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{9}$$

using (8) where

$$\text{softmax}(z)_\text{i} = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} \tag{10}$$

and the matrix operations are linear transformations like (2) [5].

The softmax function converts the rows of a matrix into a probability distribution, which represent the relative importance of tokens in the input sequence [10].

Note that taking the softmax of the linear transformation of the matrices Q, K^T is equivalent to (3). So, the query from each token is checking its similarity with every key and creating context by multiplying those similarities with the value matrix to prioritize the most similar tokens [7].

Masking Attention

Masking allows the attention mechanism to generate context at any position from previously generated output tokens while preventing the context from subsequent output tokens, i.e., tokens that have not been generated yet [5]. This is practiced in the decoder multi-head self-attention layer. The goal is to modify the calculated weighted matrix into a lower triangular matrix. Computationally

$$\text{Masked Attention}(\text{Q}, \text{K}, \text{V}) = \text{softmax}\left(\frac{QK^T + M}{\sqrt{d_k}}\right) V \tag{11}$$

where

$$M = \begin{bmatrix} 0 & -\infty & \cdots & -\infty \\ 0 & 0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & -\infty \\ 0 & \cdots & 0 & 0 \end{bmatrix} \tag{12}$$

is an $N \times N$ triangular matrix where the entries above the main diagonal are set to negative infinity as seen in (12). When (12) is applied in (11), all elements equal to negative infinity become zero, creating a lower triangular weighted matrix.

Multi-Head Self-Attention

In multi-head self-attention, each of the query, key, and value matrices are split into H even matrices to obtain the resulting tensors $Q_i', K_i', V_i' \in \mathbb{R}^{N \times \frac{d_{\text{model}}}{H} \times H}$. Each head also has its own corresponding weight matrix for the query, key, and value. This adds another dimension to the attention block, where each attention head performs self-attention to the matrix in the H^{th} dimension in parallel, producing H outputs in total. These outputs are concatenated and linearized as seen in Figure 5. Using multiple attention heads in a multi-head attention mechanism enables the model to jointly attend to information from different representation subspaces, providing a richer and more diverse set of learned relationships in the input data [5].

A tensor is a mathematical object that generalizes the concept of scalars, vectors, and matrices. Tensors can be thought of as r-dimensional arrays, where r represents the rank of the tensor [9].

Encoder-Decoder Attention

Each decoder stack has a second attention mechanism. The encoder-decoder attention mechanism is designed to capture relationships between different positions in the input sequence while generating tokens in the output sequence. The queries come from the previous decoder layer, and the keys and values come from the output of the encoder [5].

Like self dot-product attention, the queries are used to score the relevance of the keys, and the values are then weighted by these scores. This means that when generating a particular element in the output sequence, the model can consider information from all positions in the input sequence [10].

Position-Wise Feedforward Networks

A feedforward network, also known as a multilayer perceptron (MLP), consists of three types of layers: an input layer, one or more hidden layers, and an output layer. The dimension of each layer can vary, and a linear transformation is performed between each layer. Each hidden layer also applies an activation function. An activation function determines the output of a neuron given a set of inputs. The activation function introduces non-linearities to the network, allowing it to learn from complex patterns in the data [3].

The position-wise feedforward network is applied independently to each token in the sequence, making it a position-specific operation. This makes the operation N-dimensional, where each input token gets its own network. The purpose of the position-wise feedforward network is to capture and process information at each position in the input sequence separately, so each token has independent weights [5]. As seen in Figure 6, a position-wise feedforward network has one hidden layer with FFn neurons which has a non-linear activation function called the rectified linear unit

$$\text{ReLu}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{13}$$

where x is the result of (2).

Introducing non-linearity promotes generality, and the activation function prioritizes relevant features in each token [5].

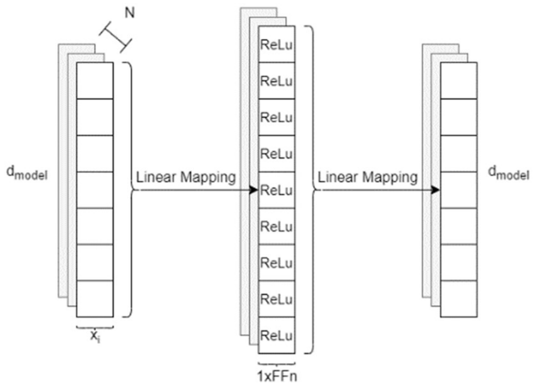


Figure 6. Position-wise feedforward network

Output Probabilities

The output of the decoder stack is mapped so that each token is linearly transformed from a d_{model} dimensional vector back to a $|V|$ dimensional vector. Each index in the vector corresponds to a distinct token in the vocabulary. (10) is applied to each of these vectors to create a probability distribution, where the value in the i^{th} position represents the probability that the i^{th} token is the output. The model expects the correct output to be the token with the highest probability in the vector.

To generate a prediction, a set of inputs and outputs $X, Y \in S$ are passed through the transformer. Each output $y \in Y$ is an $N \times |V|$ matrix where each row is one-hot encoded such that

$$y_i = [0,0,\dots,1,\dots,0,0] \in \mathbb{R}^{|V|} \tag{14}$$

representing the i^{th} row where the element in the j^{th} index is 1 and all other elements are zero [5]. Essentially, the tokens in each position in the expected sequence are known, so their probabilities are set to one.

Training with Backpropagation

During training, the weights of the embedding matrix and the linear transforms are updated through backpropagation, allowing the model to learn meaningful representations for words based on the context in which they appear [3].

Backpropagation is an algorithm used to train artificial neural networks by minimizing the error between the predicted output and the true output. It uses a cost function for the model to update the weights and biases in the model. The algorithm involves running a 'forward pass' on the network, where the network generates predictions for a training set of inputs and outputs. The algorithm calculates how well these predictions were made using a cost function, in this case the cross-entropy cost function

$$C(w,b) = -\tfrac{1}{n}\textstyle\sum_i [y_i \ln \hat{y}_i + (1 - y_i)\ln(1 - \hat{y}_i)] \tag{15}$$

where n is the number of tokens in a sequence with corresponding expected outputs y_i and predicted outputs $\hat{y}_i$. Note that (15) becomes close to zero when $\hat{y}$ is approximately y and is much larger when $\hat{y}$ is dissimilar from y. (15) is commonly used in seq2seq models as it is designed to be used for probability distributions, where the output ranges from zero to one [3].

Backpropagation works to adjust the weights after each forward pass to minimize the cost function. Recall the partial derivative of a function is the slope of the function. Intuitively, for a weight or bias parameter θ_i, the partial derivative of the cost function

$$\frac{\partial C}{\partial \theta_i} \tag{16}$$

could be used to evaluate the impact of each weight on the cost function. Modifying a weight or bias with a larger (16) will impact the cost function more dramatically than when (16) is near zero. In other words,

$$\nabla C = \langle \frac{\partial C}{\partial \theta_1}, \frac{\partial C}{\partial \theta_2}, \dots, \frac{\partial C}{\partial \theta_n} \rangle \tag{17}$$

given there are n learned parameters in the model. Thus, the model modifies the learned parameters by

$$\theta' = \theta - \eta \nabla C \tag{18}$$

where η is a small positive number called the learning rate [3]. The learning rate controls the step size during parameter updates, which is instrumental for training convergence. The idea is to decrease the cost function by incrementing η toward the minimum of the function until it is reached. A small η generally causes slow convergence, while a larger η can cause oscillations, preventing the error from dropping below a certain value [3].

To calculate (17), the chain rule is applied across each layer

$$\frac{\partial C}{\partial \theta_i} = \frac{\partial C}{\partial Z_i} \cdot \frac{\partial Z_i}{\partial \theta_i} \tag{19}$$

where Z_i is the input of a given layer. (19) can be expanded using (15) into

$$\frac{\partial C}{\partial \theta_i} = -\frac{1}{n} \sum_i \left(\frac{y}{\hat{y}_i} - \frac{1-y}{1-\hat{y}_i} \right) \cdot \frac{\partial Z_i}{\partial \theta_i} \tag{20}$$

which can further be expanded using the chain rule based on the input layer [3]. For a linear layer from (2)

$$\frac{\partial Z_i}{\partial w_i} = x_i \text{ or } \frac{\partial Z_i}{\partial b_i} = 1 \tag{21}$$

For a position-wise feedforward network from (13)

$$\frac{\partial Z_i}{\partial w_i} = \frac{\partial Z_i}{\partial x_i} \cdot \frac{\partial x_i}{\partial w_i} = \begin{cases} 0 & \text{for } x_i \leq 0 \\ x_i & \text{for } x_i > 0 \end{cases} \tag{22}$$

or (21) for biases [1]. In the attention vectors, from (9)

$$\frac{\partial Z_i}{\partial w_{i,j}} = \frac{\partial \text{softmax}\left(\frac{qk^T}{\sqrt{d_k}}\right)}{\partial \left(\frac{qk^T}{\sqrt{d_k}}\right)} \cdot \frac{\left(\frac{qk^T}{\sqrt{d_k}}\right)}{\partial w_{i,j}} v$$

given that the derivative of (10)

$$\frac{\partial \text{softmax}(x)_i}{\partial x_j} = \text{softmax}(x)_i \cdot \delta_{i,j} - \text{softmax}(x)_j$$

$$\frac{\partial Z_i}{\partial w_{i,j}} = \text{softmax}\left(\frac{qk^T}{\sqrt{d_k}}\right)_i \left(\delta_{i,j} - \text{softmax}\left(\frac{qk^T}{\sqrt{d_k}}\right)_j \right) \cdot \frac{v}{\sqrt{d_k}} \left(\frac{\partial q_i}{w_{i,j}} k_j + \frac{\partial k_j}{w_{i,j}} q_i \right)$$

where

$$\frac{\partial q_i}{w_{i,j}} k_j = \begin{cases} q_i k_j & \text{if } w_{i,j} \in k_j \\ 0 & \text{otherwise} \end{cases} \tag{23}$$

$$\delta_{i,k} = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases} \tag{24}$$

So

$$\frac{\partial Z_i}{\partial w_{i,j}} = \text{softmax}\left(\frac{qk^T}{\sqrt{d_k}}\right)_i \left(\delta_{i,j} - \text{softmax}\left(\frac{qk^T}{\sqrt{d_k}}\right)_j \right) \cdot \frac{2 q_i k_j v}{\sqrt{d_k}} \tag{25}$$

This can be applied for the queries, keys. Values are calculated like linear layers [10].

Batching the Input

Transformers can batch the input during training, meaning that a set of inputs and their corresponding outputs $B \in S$ are passed through the transformer simultaneously, where S is the set of all training data inputs and corresponding outputs. Batch processing adds a dimension to the input matrix T to form the tensor $T \in \mathbb{R}^{N \times |V| \times |B|}$. Consequently, transformer operations will now occur in $|B|$ independent dimensions simultaneously. The model computes the loss and gradients based on the predictions for each input in the batch, and then the model parameters are updated once, using the average gradient across the batch. The averaged gradient

$$\nabla C = \frac{1}{|B|} \sum_T \nabla C_T \tag{26}$$

would be used in the backpropagation for all the inputs in the batch [10].

Overfitting and Regularizations

Overfitting occurs in a neural network when the model learns not only the patterns in the training data but also adjusts to the noise and randomness of the data. This causes the network to perform worse on unseen data sets.

Dropout is a regularization technique used in neural networks to prevent overfitting. The technique mitigates overfitting by randomly setting a certain portion of token weights to zero during each training iteration. This forces the network to rely on different pathways to make predictions, developing a more robust network.

In a transformer, dropout is applied at the output of each sublayer in the encoder and decoder.

III. Approach

A transformer model, as outlined above, was implemented to perform seq2seq; given a function as an input sequence, predict the primitive, or indefinite integral as the output sequence.

Training and Evaluation

A training dataset and evaluation dataset were used from [1] which included generated equations using various methods to create a broad dataset. Note that the expressions are stored as expression trees and notated with pre-node traversal [11]. An example of this can be seen in Code Segment 1.

Code Segment 1. Example of input (line 1) and output (line 2) in the dataset

```
sub Y' mul INT+ 4 mul pow x INT+ 2 add mul INT+ 2 sin x mul x cos x

add mul INT- 8 cos x add mul INT- 8 mul x sin x add mul INT+ 4 mul pow x INT+ 2 cos x mul INT+ 4 mul
pow x INT+ 3 sin x
```

The model was run over a constant set of training data with various parameters. A set size of 6400 was used for training. An epoch is a single pass through the entire training dataset. During one epoch, the model sees each input or batch in the training dataset and updates the learned parameters accordingly. The model is trained for a set number of epochs. The model then predicts an output for each input in the evaluation dataset.

Hyperparameters

Experiments were conducted with various hyperparameters to optimize learning in the model as seen in Table 1.

Table 1. List of the modified hyperparameters to optimize model performance

Hyperparameters	Values
Number of epochs	1, 3, 5, 10
Batch size	1, 32, 64
Model dimension	64, 128
Feedforward forward size	256, 512
Number of encoder and decoder layers	4, 8
Number of attention heads	4, 8

Metrics

To determine the accuracy of the trained model, two methods were used. The token accuracy which compared each token of the generated output and expected output

$$\text{Accuracy}_{\text{token}}\left(\frac{c}{n}\right) \tag{27}$$

where c, n represent the number of correct tokens and total number of tokens respectively. The true accuracy

$$\text{Accuracy}\left(\frac{C}{N}\right) \tag{28}$$

where C, N represent the number of correctly predicted output sequences and the total number of predicted words respectively.

The model is determined to be successful if it can evaluate with a true accuracy of 80% or higher, clearly demonstrating a learned understanding of integration patterns. The token accuracy was used to observe the closeness of incorrectly generated sequences to their expected outputs.

IV. Implementation

A. The Model

The neural network model was implemented in Python 3 using PyTorch. PyTorch provides powerful computation for tensor analysis and differentiation, making it capable of computing the forward pass and backpropagation of a transformer [12].

Linear Layers and Embeddings

A PyTorch function was used to create linear layers, as seen in Code Segment 2, and embeddings, as seen in Code Segment 3.

Code Segment 2. PyTorch linear layer function

```
linear_layer = nn.Linear(input_size, output_size)
```

Code Segment 3. PyTorch embedding function

```
self.encoder_embedding = nn.Embedding(
            vocab_size,
            d_model,
            padding_token
)
```

Masking

Masks were created to pad input sequences and to maintain autoregressive nature of the model. This was done using binary operations and PyTorch as seen in Code Segment 4.

Code Segment 4. Function using PyTorch for masking the input and output sequences

```
def generate_mask(self, src, tgt):
    src_mask = (src != 0).unsqueeze(1).unsqueeze(2)
    tgt_mask = (tgt != 0).unsqueeze(1).unsqueeze(3)

    seq_length = tgt.size(1)
    nopeak_mask = (
      1 - torch.triu(torch.ones(1, seq_length, seq_length), diagonal=1)).bool()
    tgt_mask = tgt_mask & nopeak_mask

    return src_mask, tgt_mask
```

The first two lines create a binary mask where true indicates non-zero elements in the sequences. The next lines create an upper triangular matrix with ones in the lower triangle and zeros in the upper triangle. Elements in target sequence mask corresponding to the upper triangle are set to false.

Multi-Head Attention

A multi-head attention class was created, which created four linear layers, query, key, values, and output (Code Segment 5).

Code Segment 5. Linear layers in multi-head attention

```
self.W_q = nn.Linear(d_model, d_model)
self.W_k = nn.Linear(d_model, d_model)
self.W_v = nn.Linear(d_model, d_model)
self.W_o = nn.Linear(d_model, d_model)
```

The class then performed scaled dot product attention as seen in Code Segment 6.

Code Segment 6. Function to perform dot product-attention

```
def dot_product_attention(self, Q, K, V, mask=None) -> torch.Tensor:

    attention = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)

    # mask attention
    if mask is not None:
        attention = attention.masked_fill(mask == 0, -float('inf'))

    weights = F.softmax(attention.float(), dim=-1).type_as(attention)
    weights = F.dropout(weights, p=self.dropout, training=self.training)

    return torch.matmul(weights, V)
```

Position-wise Feedforward Network

Using a PyTorch function for (13), input was linearly transformed, run through the rectified linear unit, and then linearly transformed again to create the output as shown in Code Segment 7.

Code Segment 7. PyTorch position-wise feedforward network

```
lin1 = nn.Linear(in_dim, dim_hidden)
lin2 = nn.Linear(dim_hidden, out_dim)

x = lin2(F.relu(lin1(input)))
```

Encoder and Decoder

The encoder and decoder were created as objects with the appropriate sublayers initialized, and the forward pass is shown in Code Segment 8.

Code Segment 8. Decoder forward pass

```
def forward(x, enc_output, src_mask, tgt_mask):
    atten_output = self_atten(x, tgt_mask)
    x = norm1(x + dropout(atten_output))
    atten_output = cross_atten(x, enc_output, src_mask)
    x = norm2(x + dropout(atten_output))
    ff_output = feed_forward(x)
    x = norm3(x + dropout(ff_output))
    return x
```

The encoder forward pass is identical except it does not include the cross-attention sublayer.

The Model

The transformer was created as an object with the appropriate sublayers initialized, and the forward pass is shown in Code Segment 9.

Code Segment 9. Transformer forward pass

```
def forward(src, tgt):
    src_mask, tgt_mask = generate_mask(src, tgt)

    src_embedded = positional_encoding(encoder_embedding(src))
    tgt_embedded = positional_encoding(decoder_embedding(tgt))

    enc_output = src_embedded
    for enc_layer in encoder_layers:
        enc_output = enc_layer(enc_output, src_mask)

    dec_output = tgt_embedded
    for dec_layer in decoder_layers:
        dec_output = dec_layer(dec_output, enc_output, src_mask, tgt_mask)

    return output_layer(dec_output)
```

Backpropagation

PyTorch handles storing and updating the learned parameters in the model. The backpropagation algorithm can be seen in Code Segment 10.

Code Segment 10. PyTorch implementation of the backpropagation algorithm

```
self.optim.zero_grad()
# reset calculated gradients

# forward pass
src, tgt = batch
src, tgt = src.to(get_device()), tgt.to(get_device())

prob_dist = self.model(src, tgt)

# flatten the output
prob_dist_flat = prob_dist.view(-1, prob_dist.size(-1))
tgt_flat = tgt.view(-1)

# compute loss
loss = self.criterion(prob_dist_flat, tgt_flat)

# backward pass
loss.backward()
self.optim.step()
```

Tokenization

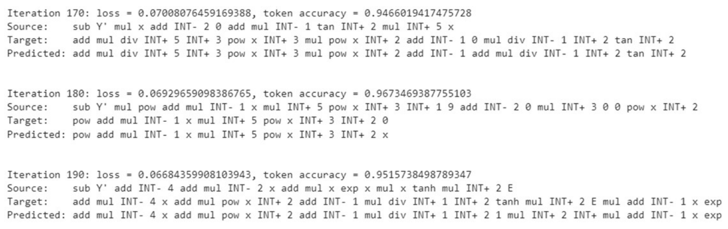
A vocabulary was created as a list of strings to encompass the tokens that the model will work with. Adapted from [1], these included a finite set of positive and negative integers and a set of unary and binary operators. To convert from string tokens into index vectors, two maps were created as seen in Code Segment 11.

Code Segment 11. Python vocabulary tokenization

```
index_to_symbol = {k: v for k, v in enumerate(self.words)}
symbol_to_index = {v: k for k, v in enumerate(self.words)}
# where k is the key for each token
# index_to_symbol[i] returns the i_th token
```

V. Results

Figure 7 shows example output from the model, showing progressive learning.



```
Iteration 170: loss = 0.07008076459169388, token accuracy = 0.9466019417475728
Source:    sub Y' mul x add INT- 2 0 add mul INT- 1 tan INT+ 2 mul INT+ 5 x
Target:    add mul div INT+ 5 INT+ 3 pow x INT+ 3 mul pow x INT+ 2 add INT- 1 0 mul div INT- 1 INT+ 2 tan INT+ 2
Predicted: add mul div INT+ 5 INT+ 3 pow x INT+ 3 mul pow x INT+ 2 add INT- 1 add mul div INT- 1 INT+ 2 tan INT+ 2


Iteration 180: loss = 0.06929659098386765, token accuracy = 0.9673469387755103
Source:    sub Y' mul pow add mul INT- 1 x mul INT+ 5 pow x INT+ 3 INT+ 1 9 add INT- 2 0 mul INT+ 3 0 0 pow x INT+ 2
Target:    pow add mul INT- 1 x mul INT+ 5 pow x INT+ 3 INT+ 2 0
Predicted: pow add mul INT- 1 x mul INT+ 5 pow x INT+ 3 INT+ 2 x


Iteration 190: loss = 0.06684359908103943, token accuracy = 0.9515738498789347
Source:    sub Y' add INT- 4 add mul INT- 2 x add mul x exp x mul x tanh mul INT+ 2 E
Target:    add mul INT- 4 x add mul pow x INT+ 2 add INT- 1 mul div INT+ 1 INT+ 2 tanh mul INT+ 2 E mul add INT- 1 x exp
Predicted: add mul INT- 4 x add mul pow x INT+ 2 add INT- 1 mul div INT+ 1 INT+ 2 1 mul INT+ 2 INT+ mul add INT- 1 x exp
```

Figure 7. Example output from the model

Training Parameters

The model was run on different epochs and with different batch sizes as seen in Table 2 to determine the optimal training conditions. The model parameters were kept constant during this test as seen in Table 3.

Table 2. Impact of number of epochs on model performance

Batch Size	Epoch	Model Accuracy	Run Time (min : sec)
1	1	0.9998	9 : 25
16	1	0.0027	3 : 40
16	3	0.5044	5 : 01
16	5	0.9455	7 : 36
32	1	0.0000	3 : 36
32	3	0.1318	5 : 24
32	5	0.4913	7 : 17
64	10	0.4902	11 : 27

Table 3. Constant model parameters used in testing

Parameters	Values
Model dimension	64
Feedforward forward size	256
Number of encoder and decoder layers	4
Number of attention heads	4
Dropout rate	0.1

Model Parameters

The model was run with different hyperparameters to evaluate their impact on performance. The training was run with batch sizes of 16 and 5 epochs, based on the results in Table 2. One hyperparameter was varied, while others were kept according to Table 3, and redundant tests are removed. The results are shown in Table 4.

Table 4. Impact of model parameters on performance

Parameters	Values	Model Accuracy
Model dimension	128	0.9982
Feedforward forward size	512	0.9747
Number of encoder and decoder layers	8	0.9855
Number of attention heads	8	0.9434

VI. DISCUSSION

The implemented transformer model demonstrates significant success in predicting primitives given input mathematical expressions. The training and evaluation datasets, sourced from [1], provide a diverse set of equations, and the model performed well above the target metric with various configurations.

Interpretation of Results

Table 2 illustrates the impact of the number of epochs and batch size on model performance. Notably, lower batch sizes and higher epochs generally lead to improved accuracy. This suggests that allowing the model more exposure to the training data results in better learning. However, there is a diminishing return, as seen in the increased runtime with higher epoch values.

Table 4 explores the influence of varying model parameters on performance. Notably, increasing the model dimension, feedforward size, and the number of encoder and decoder layers generally leads to improved accuracy. However, these improvements come at the cost of increased computational complexity and training time.

Implications of Findings

The high accuracy achieved by the model suggests that the transformer has effectively learned patterns associated with integration. This implies that this method could be used for solving complex integration problems efficiently in the place of existing computational software.

Potential Extensions

It is important to test the model's ability to generalize to different mathematical expressions by incorporating diverse sets of integration problems. The current study primarily focuses on a broad range of equations generated through various methods, providing a foundation for understanding the model's capabilities. However, further investigation into specialized integration tasks, involving more intricate functions or specific mathematical domains, could provide an evaluation of the model's robustness and adaptability.

VII. CONCLUSION

In summary, this study presents a transformative approach to predicting primitive or indefinite integrals using a transformer model. The model exhibits commendable accuracy, especially when trained with optimal hyperparameters, showcasing its ability to discern complex integration patterns. The research highlights the impact of key parameters on the model's performance, emphasizing a delicate balance between accuracy and computational efficiency. The findings have significant implications for symbolic mathematics and computational algebra, offering a valuable tool for problem-solving.

VIII. References

[1] G. Lample and F. Charton, "Deep Learning for Symbolic Mathematics," arXiv:1912.01412, 2019.

[2] S. Stenlund, The origin of symbolic mathematics and the end of the science of quantity, Sweden: Department of Philosophy, Uppsala University, 2014.

[3] M. A. Nielsen, Neural networks and deep learning, San Francisco, CA, USA: Determination press, 2015.

[4] K. R. Chowdhary, "Natural Language Processing," in *Fundamentals of Artificial Intelligence*, Jodhpur, Rajasthan, India, Springer New Delhi, 2020, p. 716.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems,* vol. 30, p. 11, 2017.

[6] Y. Jia, "Attention mechanism in machine translation," *Journal of physics: conference series,* vol. 1314, p. 7, 2019.

[7] PyTorch, "Word Embeddings: Encoding Lexical Semantics," 14 September 2021. [Online]. Available: https://github.com/pytorch/tutorials/blob/main/beginner_source/nlp/word_embeddings_tutorial.py.

[8] T. Mikolov, K. Chen, G. Corrado and J. Dean, "Efficient Estimation of Word Representations in Vector Space," Proceedings of Workshop at ICLR, 2013.

[9] L. Brand, Vector & Tensor Analysis, Mineola, New York: Dover Publications, Inc., 2020.

[10] R. E. Turner, "An Introduction to Transformers," Department of Engineering, University of Cambridge, UK, Microsoft Research, Cambridge, UK, 2023.

[11] D. W. Harder, "Binary Trees," in *ECE 250 Algorithms and Data Structure.*, Department of Electrical and Computer Engineering, University of Waterloo, 2011, p. 8.

[12] A. Paszke, S. Gross, F. Massa, A. Lerer and J. Bradbury, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, vol. 32, Curran Associates, Inc., 2019, pp. 8024--8035.