UMEÅ UNIVERSITY
Department of Physics
At Saab Training Systems

July 5, 2023

Efficient use of resources when implementing machine learning in an embedded environment

Author: Johannes Eklöf

Abstract

Machine learning and in particular deep-learning models have been in the spotlight for the last year. Particularly the release of ChatGPT caught the attention of the public. But many of the most popular models are large with millions or billions of parameters. Parallel with this, the number of smart products constituting the Internet of Things is rapidly increasing. The need for small resource-efficient machine-learning models can therefore be expected to increase in the coming years. This work investigates the implementation of two different models in embedded environments. The investigated models are, random forests, that are straight-forward and relatively easy to implement, and transformer models, that are more complex and challenging to implement. The process of training the models in a high-level language and implementing and running inference in a low-level language has been studied. It is shown that it is possible to train a transformer in Python and export it by hand to C, but that it comes with several challenges that should be taken into consideration before this approach is chosen. It is also shown that a transformer model can be successfully used for signal extraction, a new area of application. Different possible ways of optimizing the model, such as pruning and quantization, have been studied. Finally, it has been shown that a transformer model with an initial noise-filter performs better than the existing hand-written code on self-generated messages, but worse on real-world data. This indicates that the training data should be improved.

Supervisors
Andreas Månsson, Adrian Hjältén
andreas.mansson3@saabgroup.com, adrian.hjalten@umu.se

Contents

Nomenclature

Notation	Meaning
θ	A split of data in a decision tree node
Q_m	Data in a decision tree node
$H()$	(Gini) Impurity function
Q	Queries
K	Keys
V	Values
w	Weight matrix of attention head
W^x	Projection matrix in attention block
μ	Mean value of activations in layer
σ^2	Variance of activations in layer
γ	Scaling matrix of normalization layer
β	Bias matrix of normalization layer
α	Step-size in gradient descent
L	Loss of a loss-function

Acronyms

Abbreviation	Meaning
GPT	Generative pre-trained transformer
AI	Artificial intelligence
NLP	Natural language processing
CART	Classification and regression trees
ReLU	Rectified linear unit
BERT	Bidirectional encoder representations from transformers
MC	Memory consumption
MS	Model size
GSL	GNU Scientific Library
NF	Noise filter
SF	Specific filter
CPU	Central processing unit
FLOPS	(singel-precision) floating point operations per second

1 Introduction

Machine learning has emerged as a powerful tool in the field of artificial intelligence and has been successfully applied in various applications such as image recognition, natural language processing, and predictive analytics. However, as the demand for real-time processing and low power consumption increases, there is a growing need for machine-learning models to run efficiently in embedded environments. In this paper, we will discuss the challenges and opportunities associated with deploying machine learning algorithms in embedded systems and explore the various techniques that can be used to optimize the performance of these algorithms in resource-constrained environments. We will also examine some of the current applications of machine learning in embedded systems and discuss the future potential of this technology.

The paragraph above was generated by ChatGPT [1] as an answer to the request: "Write a short introduction to a paper on machine learning in embedded environments.". It is amazing how fast this technology, that seemed to be many years into the future just a year ago, has now become a well-known tool for people across ages and professions, reaching 100 million users in just two months [2]. The reason to start the introduction with an AI-generated paragraph is, however, more specific than just showing what AI and machine-learning models are capable of. The capital T in ChatGPT (Chat generative pre-trained transformer) stands for Transformer, a type of NLP (natural language processing) model that sparked the NLP revolution when introduced in 2017 [3], and the transformer model happens to be one of the two models that this paper will investigate and revolve around.

But unlike ChatGPT that has 175 billion parameters and costs more than $4 million to train [4], the transformers in this study have to operate under much harder constraints when it comes to memory and energy usage. Fortunately the investigated task is more restricted and should not be anywhere near as demanding as the tasks given to ChatGPT.

As promised in the first paragraph this paper will focus on the challenges and possibilities associated with machine learning in embedded environments. A particular focus will be given to the process of training models in a high-level language on a platform with access to relatively powerful computational resources and then implementing the forward pass or predicting part of the trained model in a low-level language on a platform where the computational power and available memory is more restricted.

1.1 Problem description

The work has been conducted at Saab AB, Training and Simulation (Saab), a branch of the defence company that focuses on solutions for simulating combat in a realistic way, with tools for executing, supervising, and evaluating military exercises performed in the field. In several of Saab's products the need for wireless communication between different units arises, and often these units are at a distance from each other. The units are then made to communicate with pulses where the information is encoded in the time difference between the pulses. Every time difference is sent multiple times with successive pulses, to make the code more robust to disturbances in the form of noise and lost pulses. A message consists of several different time differences sent according to certain rules.

A general problem with communication is that the signal can pick up noise on its way from one

unit to another. A noise pulse can split an existing time interval in two or or pulses can get lost on the way resulting in a doubling of the corresponding time intervals. Or the receiving unit can be exposed to several codes at once, or codes from another source, in such a way that the time intervals of the different codes split each other and the codes become superimposed.

In Figure 1, an illustration of what such a message might look like at the sending unit and at the receiving unit after it has been exposed to noise and other disturbances, and lost pulses on the way, can be seen. The numbers in the code are the time intervals between the pulses, measured in time units, tu.

Figure 1: An example of a message at the sending unit (top), and at the receiving unit (bottom). The coloring of the numbers (time intervals) is there to highlight where the time intervals from the original message appears in the noise affected codes. In this example the codes are self-generated and the noise and the distortions have been artificially added. The actual numbers have been blurred out on request by Saab.

The codes can become quite distorted and complex to decode, therefore some type of program is needed to decode the message, or in other words, to extract the pulses that are part of the original code and combine them together into a final message. Today this decoding is done with human written code following explicit conditions.

The scope of the study is to investigate if some or all of the deciphering can be done by a machine-learning based model, and study if this can lead to an increased robustness against noise and distortions of the codes. Hopefully, the model can learn to detect patterns that are hard to manually describe in an algorithm, leading to a better capability when it comes to classifying noise and decoding messages. Another benefit of using a machine-learning approach in decoding the message is that in the case of a major change to the conditions in the environment where the decoding happens, data representing the new conditions could be gathered and the model could simply be retrained on this new data, instead of having to rewrite the algorithm.

If the machine-learning approach proves successful, the next step is to implement the already trained model in an embedded environment, where the demands on low memory usage and energy efficiency are higher. This also requires a translation to a low-level language, such as C.

One approach to the problem that is investigated, and which, as far as we know, has not been

tried before, is to see the decoding as a translation from the distorted and corrupt code to another language representing the actual message. Because of this, an NLP model in the form of a transformer model is evaluated on solving the problem. The other major type of model being evaluated in this paper is random forests, a type of model mainly used for classification and regression tasks. The typical random forest requires far less computations than the typical transformer, and is also less complex when it comes to what happens when the model makes a prediction. The idea of having two types of models, one complex and one simple, is to compare and discuss machine learning on embedded platforms from different points of view. Touching upon what is possible today, what could be done tomorrow if the capacity and efficiency of the platforms were improved, and if it is really necessary to use large and complicated models.

2 Theory

2.1 Random Forest

The roots of the random forest algorithm can be traced all the way back to 1936 when Ronald Fischer published his paper on discriminant analysis. In the paper he established the foundations for multivariate data analysis by using the petal- and sepal width of Iris flowers to determine their specific species [5]. The data set he studied is still often used to introduce new students to machine learning, and the discriminant analysis methods he developed have been a precursor to modern-day decision trees.

The work of many people over many years in developing, improving and implementing this discriminant analysis on computers for use in data analysis, eventually led to the idea of stacking several decision trees together to get better predictions. In 1995 the first random forest algorithm was introduced by Tin Kam Ho [5].

A decision tree is an algorithm of nested if-statements in a tree structure, commonly used for classification and regression tasks in machine learning. It consists of an initial root node with branches going out to internal nodes, its children nodes. The internal nodes in turn have branches going out to its children nodes, until a leaf node is reached where a decision is taken, for example on the most probable classification [6]. Each node contains a condition used to determine which child node should be visited next when using the tree to make a decision. The most common way of branching a node is with a binary split that creates two child nodes.

In order to get more accurate predictions in a classification task the decision tree can be grown with more nodes, enabling a finer split. But eventually this leads to a risk of overfitting, and to prevent this several trees can be used together to make the prediction, this is what is called a Random Forest [6]. The trees are trained either on a random subset of the data or only using a random fraction of the available features of the data, or both. In the following subsection we will study the specific random forest algorithm used in this project: scikit-learn's version designed for Python [7].

2.1.1 Decision Tree

When using a decision tree in a classification task, the data is sent to the root node which has a condition, consisting of a feature to examine and a value or state to compare against. The data is then divided according to the value or state of this feature for each data point. The divided parts are then sent to different child nodes (down different branches of the tree), where the process is repeated until all data points have reached a leaf node and a classification have been made of each data point.

Scikit-learn's decision tree uses an algorithm known as CART (Classification and Regression Trees). It is able to handle numerical features and target variables, however in this work only categorical target variables have been used. In scikit-learn's optimized version only numerical features are allowed [8], however, a categorical feature can easily be converted to a numerical one by representing each category with a number.

A CART decision tree is trained with a greedy algorithm that makes the split, $\theta = (j, t_m)$ based on a feature j and a threshold t_m, that is optimal right there in the node, resulting in a locally optimal solution but most often not a globally optimal one. The goal of the split is to group samples from the same category in the same group, in other words, divide the data Q_m at node m into $Q_m^{\text{left}}(\theta)$, that goes to the left child node, and $Q_m^{\text{right}}(\theta)$, that goes to the right child node in the best possible way. To measure how good a split is, an impurity function $H()$ is used. The split that is chosen is the one that minimises the function

$$G(Q_m, \theta) = \frac{n_m^{\text{left}}}{n_m} H(Q_m^{\text{left}}(\theta)) + \frac{n_m^{\text{right}}}{n_m} H(Q_m^{\text{right}}(\theta)), \tag{1}$$

where n_m stands for the number of samples in the data at the node m, or in the left or right group after the divide [9].

The impurity function used in this work is called Gini impurity and it measures the probability of wrongly classifying a random sample from the data in the node [6]. If the proportion of data points from class k in node m is denoted p_{mk}, then the Gini impurity function is given by [9]:

$$H(Q_m) = \sum_k p_{mk}(1 - p_{mk}) \quad . \tag{2}$$

This function is then applied to each group of the split to measure the impurity of the group. The process of branching nodes continues until one of the three following criteria has been met: all leaves are pure, meaning that they only contain data of one specific class; the $min\_samples\_split$ has been reached, specifying a minimum number of samples in the node to allow a split; or the max_depth has been reached, specifying the maximum distance from the node to the root node [8].

If the tree is to be trained on data where one or several of the classes contain significantly more samples than the rest, a balanced class weight can be used that gives the classes weights that are inversely proportional to the number of samples in the class. The weight of the class is then multiplied with the impurity of each class, before deciding the optimal split. Or if the data contain classes that are more important to classify correctly a higher weight can be given manually to those classes.

2.1.2 Ensemble of Decision Trees - A Random Forest

To avoid overfitting the decision tree classifier, an ensemble of several trees can be used. A common ensemble is a random forest. The goal is to get the prediction errors of the trees to become uncorrelated and cancel each other, reducing the overfitting of the model. This is done by only giving each tree a random subset of the full data set to train on, determined by the variable $max\_samples$, or only access to a random subset of the features, determined by the variable $max\_features$, or both [8]. Both the subset of the data and the subset of the features are drawn with replacement.

When making a classification with a random forest, the prediction of each tree is given by a vector containing the proportion, or the weighted proportion if weights have been used, of each class of the data reaching the classifying node during training. The predictions from all the trees are averaged to give a final prediction from the whole tree.

2.2 Transformer

The transformer model was first introduced in the paper "Attention is all you need" by Vaswani et al. in 2017 [3]. It is primarily designed for translation and text generation, a so-called sequence-to-sequence model. It consists of a layered structure of neural networks that together forms an encoder and a decoder. The encoder takes a sequence of numbers, for example word tokens representing certain words in a vocabulary, and encodes them into a new sequence of numbers. This sequence is supposed to convey the meaning of the input sentence to the decoder. The decoder then gets this sequence, together with any previously translated tokens, as input and generates a decoded sequence. This decoded sequence can, for example, consist of word tokens representing words in a new language.

What is special with the transformer is that it does not rely on recurrent neural networks to handle a sequential input, but rather uses something called attention, or more specifically multi-head attention. This allows the transformer to process an entire input sequence simultaneously and let the elements in the input influence the meanings of each other.

The simultaneous processing of the input means both that calculations of predictions and gradients can be computed in parallel over the whole sequence and that the same weight matrices are not multiplied together once for each element in the input, as it is in a recurrent neural network, possibly leading to exploding or vanishing gradients. This in turn lets the transformer train a lot faster than previous sequence-to-sequence models, and makes it possible for it to learn relationships between words over longer distances in a sequence [3].

My implementation of the transformer builds on TensorFlow's variant [10] of the original transformer. In Figure 2, a sketch of the transformer from the original paper is shown, the left half is the encoder part and the right half is the decoder.

The figure shows how the input sequence first gets embedded with both word embedding and positional embedding in an attempt to give the encoder more information to use when encoding the meaning of the input sequence. The input then goes through N encoder layers, each with a multi-head attention block that looks at the internal relationships between the tokens in the sequence to understand its meaning, and a feed-forward layer that sends each token through

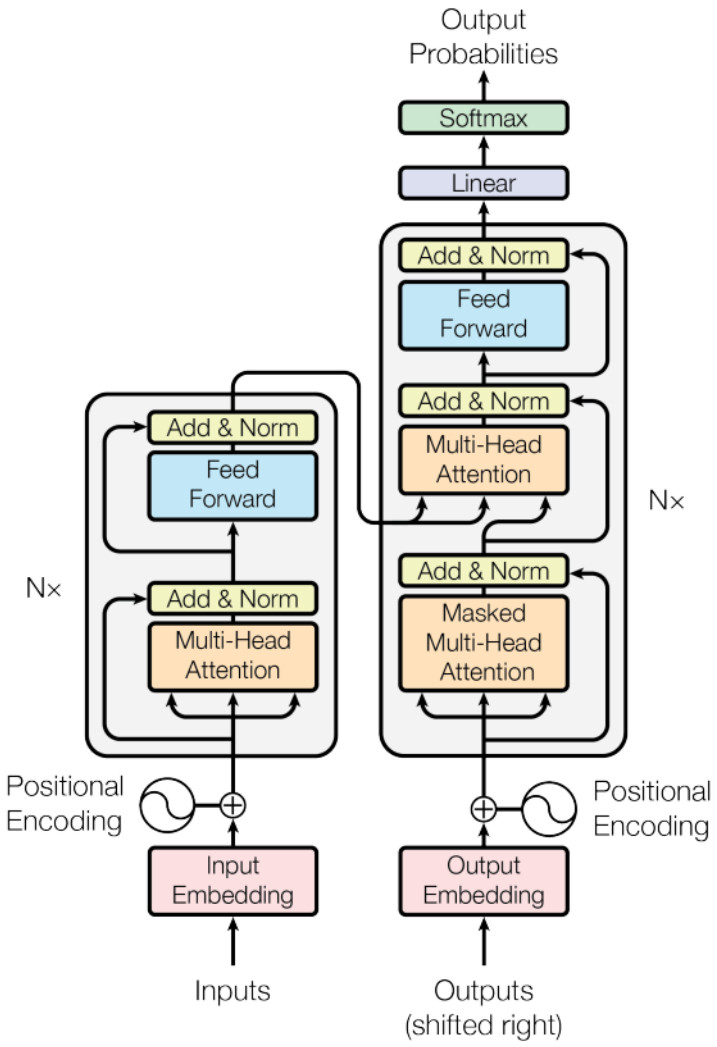

Figure 2: The transformer, taken from the original paper [3]. The left part shows one of the N encoder layers and the right part shows one of the N decoder layers. The arrows show how the activations travel through the different blocks of the transformer, where each attention blocks takes three inputs: keys, values, and queries.

the same neural network which makes the input to each layer differ more, because of the non-linearity in the layer. Thus we can get the attention blocks to perform slightly different tasks. This layer also allows the different heads to interact. Between and after these blocks, add- and normalization layers are added to make the training of the model faster and more efficient.

The decoder input consists during training of the embedded target sequence and during inference of any previously translated tokens which are also embedded. This input is fed through the N layers of the decoder. The first part of such a block is a masked self-attention block. It is followed by an attention block that combines the output from the first attention block with the output from the last encoder layer. This attention block establishes the connection between the input sequence and the target sequence. The last block is a feed-forward block and add- and normalization layers between the blocks are present in the decoder as well. After the last decoder layer, a linear layer and a preceding softmax layer are applied to turn the decoded sequence into next-token probabilities.

2.2.1 Multi-headed attention

The need for attention in sequence-to-sequence models arises because the underlying structures of the language of two sequences can be different, meaning that the second word in the input sequence might not be the most important for choosing the second word in the output sequence. Therefore, some kind of method for directing where in the input sequence the model should focus its attention when choosing the next token in the output sequence is beneficial. The attention model described here uses three types of inputs, denoted as: queries, keys, and values. It can be described as a kind of dictionary where every key has a matching value and the query is compared to the keys to find a mix of corresponding values. If the queries, keys and values are all the same, the model becomes a self-attention model capable of encoding the meaning of an input sequence.

The multi-heads enter through a number of different projection matrices that the queries, keys and values are projected through, the number is determined by the variable num_{heads}. The idea is that each head can attend differently to the word embeddings at different positions, thus focusing on different aspects of the meaning of words when translating different parts of the sequence.

The model receives a set of queries, Q, keys, K, and values, V, as input. They come in the form of matrices with *(batch size, input sequence length, d_{model})* as dimensions. d_{model} is the dimensionality of the embeddings and the size of the last dimension of the output of each layer. They are then projected through N parallel fully connected linear neural networks, where N is the number of heads in the multi-headed attention block. After the projection the dimensions become *(batch size, input sequence length, N, d_{model})*.

To find out how the queries correspond to the key-value pairs, the dot product is taken between the query matrix and the transposed key matrix. The product is then scaled by dividing with $\sqrt{d_{model}}$ to avoid that the dot product becomes too large for large d_{model}, since this could cause the gradient of the following layer to vanish [3]. The product is then run through a softmax layer, which is a function that for every element takes the exponent of that element and divides it by the sum of all exponents over each word (over the last dimension of the product matrix). The softmax is thus a form of normalization that enhances the differences between the elements.

The resulting matrix is called the weight matrix and the output is the weighted sum of the values using these weights, mathematically the attention mechanism is described as

$$\text{Attention}(Q, K, V) = \frac{e^{w}}{\sum_{j=1}^{d_{model}} e^{w_j}} V, \tag{3}$$

where w is the weight matrix,

$$w = \frac{QK^T}{\sqrt{d_{model}}}.$$

Finally the attention output of the different heads is concatenated and run through a last fully connected linear layer, producing an output with dimensions *(batch size, query sequence length, d_{model})*, which are the same dimensions as the query input.

The parameters that the attention model has to learn in order to generate meanings and relationships between the tokens are the weights and biases of the three by N projection matrices and the weights and biases of the final output layer.

The entire multi-head attention function is described by:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_N)W^O, \tag{4}$$

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

W_i^Q, W_i^K, W_i^V, and W^O are trainable parameter matrices that are $\in \mathbb{R}^{d_{model} \times d_{model}}$ [3].

2.2.2 The encoder

The encoder has a sequential structure with N identical encoder layers after each other, shown in the left half of Figure 2.

An encoder layer consists of a self-attention block that gets its input from the previous layer, or gets the embedded input sequence as input in the case of the first layer. The queries, keys and values of this block are all the same. The next part of the layer is a block with a residual connection to the input of the attention block. The residual connection is there to make training of the deep networks more efficient [11], it adds the input of the attention block to its output before sending it on. This makes the default of the network before training more like the identity matrix instead of the zero matrix, since the weights of the network are initialized around zero. Thus, the network learns to adjust the signal rather than creating a totally new representation of it.

Following the residual connection comes a layer-normalization layer. In layer normalization the mean, μ, and variance, σ^2, is calculated for each feature, in our case across the embedding dimension of each token. Every element in the input matrix is then normalized by taking the difference between the element and its corresponding mean and dividing by the square root of its corresponding variance plus a small number ϵ for numerical stability. This is then scaled by multiplying with a matrix γ of learned weights and shifted by adding another matrix with trainable parameters β. All in all this becomes

$$\text{LayerNormalization}(M) = \frac{(M - \mu)}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta, \tag{5}$$

where μ and σ^2 takes the form of matrices, with one tensor-dimension less than the matrix of activations M.

The purpose of this layer is to speed up training by stabilizing the back propagation of the gradients [12]. Because the normalization makes the activations (input matrices) have the same magnitude in each layer the gradient can influence all trainable parameters more evenly.

The next part of the encoder layer is the feed-forward layer, which is there both to introduce non-linearity in the model through the ReLU layer, and to enable the attention heads to interact.

The feed-forward layer consists of two fully connected linear networks with a ReLU activation (if an element is negative put its value to zero) between them. This is described by:

$$FF(x) = \max(0, xW_1 + b_1)W_2 + b_2[3], \tag{6}$$

where W_1 and b_1 are trainable weights and biases that have a last dimension of d_{ff} and W_2 and b_2 are also trainable weights and biases that have a last dimension of d_{model}, such that both input and output of this layer have the same dimensionality, d_{model}.

The final part of the encoder is another residual connection that is added with the output from the feed-forward layer and then run through a layer-normalization layer. The output is then sent to the next encoder layer in the encoder, or for the last layer, sent to the decoder.

2.2.3 The decoder

The decoder is more complex than the encoder with three major blocks instead of two. It is shown to the right in Figure 2. The decoder also consists of N identical layers, where the first layer takes the embedded target sequence as input during training and the previously predicted part of the translation as input during inference.

The first part of the layer is a multi-head attention block in which a mask has been applied in the dot product of the queries and keys, such that positions in the sequence cannot attend to tokens at later positions. This prevents the decoder from looking at the next part of the target sequence when making a prediction during training. The mask sets the values that are supposed to be masked to $-\infty$.

The output from the first attention block is added with a residual connection and run through layer normalization. The normalized output is then sent as the query to the next attention block, while the key-value pairs of this block come from the output of the last encoder layer. This lets the decoder attend to the whole encoded input sequence, and use this together with the previously predicted sequence to predict the next token.

The output is once again added with a residual connection that is the same as the queries that went into the attention block and layer normalization is applied. Then the final part of the decoder consists of a feed-forward block with the same architecture as in the encoder, with new trainable weights for every layer here as well.

After a residual connection and a final layer normalization, the output is sent to the next layer in the decoder or, for the final layer, to a linear layer followed by a softmax layer to make the prediction of the next token in the translated sequence.

2.2.4 Architecture of the entire model

The architecture of the entire model is shown in Figure 2. In addition to the encoder and decoder that has already been described, the transformer applies word embeddings and positional encoding to both the input sequence and the target sequence, see below. The input to the encoder then has the dimensions *(batch size, input sequence length,* d_{model}*)* and the input to the

decoder *(batch size, target sequence length, d_{model})*.

After the last decoder layer, the output is sent through a linear layer with trainable parameters that transform the output to have dimensions *(batch size, target sequence length, target vocab size)*, where *target vocab size* is the number of different tokens in the vocabulary of the target language. This means that for every position in the target sequence, the model predicts how likely each token in the target vocabulary is to fill that position.

The final softmax layer is then added to transform the likelihoods to numbers between 0 and 1 for each token, in a way such that the total likelihood over all tokens sums to one.

2.2.5 Word embeddings

In order for the transformer model to learn more complex meanings of the words or tokens, some extra information is needed. This information is given to the transformer in the form of word embeddings, which are unique vectors for each token in the vocabulary, with a fixed number of dimensions. They are randomly initialized at first and later learned by the transformer during training. The idea is that words with similar meanings become mapped to positions that are close to each other in embedding space. The self-attention mechanism of the transformer can then use these embeddings to let the meanings of the words in the sentence influence each other. For example in the meaning "Let the bank of the river flourish.", the position of the word bank is shifted from a position related to money in embedding space to a position more closely connected to water by the word river.

In this model the word-embedding vectors are multiplied by the square root of their dimensionality to give them a suitable scaling compared to the positional embedding [10].

2.2.6 Positional encoding

The attention-layers have no way of knowing in which order the input tokens were sent into the transformer. All the nodes corresponding to the tokens just light up all at once. But since the meaning of a sentence varies a lot depending on the order of the words, we need a way of conveying the positional information of the sequence. This is done by adding a positional embedding to the ordinary word embeddings. The two embeddings have the same size and the idea is that the positional encoding shifts the position of each token in embedding-space slightly depending on its position in the sequence. This is done by alternating sine- and cosine waves with decreasing frequency according to

$$\begin{aligned} PE_{(pos,2i} &= \sin(pos/1000^{2i/d_{model}}) \\ PE_{(pos,2i+1} &= \cos(pos/1000^{2i/d_{model}}), \end{aligned} \tag{7}$$

where pos is the token's position in the sequence, i is the dimension in the embedding, and d_{model} is the total number of dimensions [3].

2.2.7 Training the transformer

During the training of the transformer, dropout layers are used in several places with a dropout rate of 0.1. A dropout layer is a layer that randomly sets some of the elements of the activation matrix (the matrix sent through the different layers) to zero, with a specified dropout rate. The rest of the elements are then scaled up by 1 / (1 - dropout rate) to preserve the total sum of the activations. The reason for adding this is that it has been shown to prevent overfitting since dropout reduces the co-adapting between different layers [13]. In the transformer, dropout layers are present in the attention blocks, between the softmax layer and the layer where the weighted sum of the values is calculated. It is also present after the last layer in the feed-forward block and after the positional encoding of both the input sequence and the target sequence.

The transformer is trained using the Adam optimizer, which is an evolved form of stochastic gradient descent introduced by Kingma and Ba in 2015 [14]. It has been developed to get a smoother descent towards the local minima of the loss function by adjusting the step size for every variable that is being updated depending on the first moment (exponentially decaying previous gradients) and the second moment (exponentially decaying previous squared gradients). The formula for Adam optimizing (giving a new value to) the variable x at time step t with the gradient $g(t) = f'(x(t-1))$ is the following:

$$x(t) = x(t-1) - \alpha(t)\frac{m(t)}{\sqrt{v(t)+\epsilon}}, \tag{8}$$

$$\text{where } \alpha(t) = \alpha\frac{\sqrt{1-\beta_2(t)}}{1-\beta_1(t)},$$

$$\text{and } m(t) \text{ is the first momentum: } m(t) = \beta_1 m(t-1) + (1-\beta_1)g(t),$$

$$v(t) \text{ is the second momentum: } v(t) = \beta_2 v(t-1) + (1-\beta_2)g(t)^2,$$

$$\text{and: } \beta_1(t) = \beta_1^t \text{ and } \beta_2(t) = \beta_2^t.$$

$m(0)$ and $v(0)$ are both initialized to zero and ϵ, the numerical stability term in the square root; β_1, the decay factor for the first moment; and β_2, the decay factor of the second moment; are all hyperparameters with values set to: $\epsilon = 1 \cdot 10^{-9}$, $\beta_1 = 0.9$, and $\beta_2 = 0.98$. The learning rate, or step size, α is made to depend on the number of steps taken during training, $step\_num$, following the formula:

$$\alpha(step\_num) = d_{model}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5}). \tag{9}$$

This was taken from the original transformer [3], with $warmup\_steps = 4000$, and corresponds to a linearly increasing learning rate for the first warm-up steps and then a decaying step size proportional to the inverse square root of the number of steps taken.

The loss function that is used to update the variables in each step is sparse categorical cross-entropy, a cross-entropy loss function used when the labels of the classes are represented by integers. The cross-entropy loss is calculated by

$$L_{\text{CE}} = -\sum_{i=1}^{n} t_i \log(p_i), \tag{10}$$

where n is the number of classes, t_i is the truth label of the ith class, and p_i is the softmax probability for the class [15].

2.2.8 Evaluation

The transformer is evaluated with a masked accuracy that feeds the model with the input sequences of the validation data and compares the corresponding predictions with the target sequences. As the target sequences can be zero-padded to match the sequence length of their specific batch, only the predictions where the corresponding labels are nonzero are counted in the accuracy calculation. The accuracy score is simply the percentage of the nonzero token labels that have a matching prediction.

2.3 Literature study

The focus of the literature study has been on optimizing deep-learning models in order to have them fit and perform on embedded devices. This is a research topic that has gained a lot of traction in the last couple of years, as state of the art models seem to grow ever larger with more and more parameters.

The paper *Efficient Deep Learning: A Survey on Making Deep Learning Models Smaller, Faster, and Better* [16] from 2021 becomes a staring point for the investigations on optimizing the transformer model. In the paper Menghani presents several approaches for making deep learning models more efficient. He begins by differentiating between inference efficiency, such as the number of parameters, RAM consumption during inference, and inference time; and training efficiency, which is for example training time, memory requirements during training, and amount of data needed for training to a certain accuracy. In this project, the data is generated artificially, and there are ample resources available during training. Therefore only the inference efficiency is of interest to us.

To reduce the size of the transformer, pruning, removing, certain parts of the model, for example a fraction of the weights in the neural networks, is a good candidate. Menghani describes the theoretical promises of pruning, but he makes the important distinction between structured and unstructured pruning. In the former, weights are removed all over the network, depending on, for example, magnitude, while in the latter, weights that belong in a block are removed together, for example by removing a parameter as input to a network and by extension, all connections belonging to that first neuron associated with the parameter. While most research have been done on unstructured pruning, it is not clear how these pruned networks should lead to smaller model size and faster inference time, without the use of specialised software or hardware, that

leverages sparse matrices. The connection is more straightforward when it comes to structured pruning, something to keep in mind when pruning the transformer.

Another method for reducing model size, and possibly also inference time, discussed in the paper [16], is quantization, where the precision of the weights and activations is reduced by quantizing to a datatype with lower precision, such as 8-bit fixed-point integers. If the model size is the only constraint, weight quantization is enough, but if the inference time is to be decreased, quantization of the activations is also necessary, so that all operations in the model happen in fixed-point space. Quantization of the activations also reduces working memory consumption, which can be beneficial. A quantization with b-bit fixed-point integers is done by first mapping 0.0 to a fixed-point value x_{q_0} and then mapping $x_{min} - \epsilon$ to $-2^{\frac{b}{2}} - 1$, keeping ϵ as small as possible and in the same way mapping $x_{max} + \epsilon$ to $2^{\frac{b}{2}} - 1$. To quantize a weight x, the following formula is applied:

$$\text{quantize}(x) = x_q = \text{round}\left(\frac{x}{s}\right) + x_{q_0},$$

where $s = \frac{x_{max} - x_{min}}{2^b}$ is the floating-point $scale$ value, an illustration of quantization is shown in Figure 3.

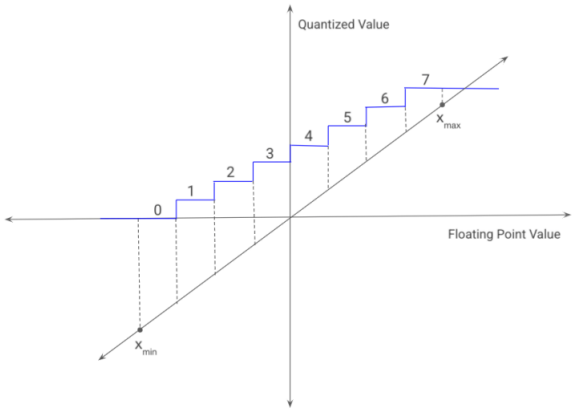

Figure 3: A schematic of a fixed point implementation with 3 bits. Source: [16]

.

Quantization of both the activations and the weights were tried in this project to get a faster and smaller transformer.

A third method to get a smaller model, that is not tried in this work, but still deserves to be mentioned because it is interesting for future studies is something called student-teacher distillation. A new much smaller model can be trained by the large model in this student-teacher distillation. During training on a classification task, the probabilities for each classification are gathered from the last layer in the large model, and the smaller model gets a new loss function that combines the usual cross-entropy loss based on the ground truth labels and a cross-entropy loss that compares the output probabilities from the small model with those from the large. Mathematically the loss L is given by:

$$\begin{aligned} L &= \lambda_1 \cdot L_{ground-truth} + \lambda_2 \cdot L_{distillation} \\ &= \lambda_1 \cdot \text{CrossEntropy}(\boldsymbol{Y}, \boldsymbol{Y^{(s)}}) + \lambda_2 \cdot \text{CrossEntropy}(\boldsymbol{Y^{(t)}}, \boldsymbol{Y^{(s)}}), \end{aligned} \tag{11}$$

where CrossEntropy is the cross-entropy loss function that takes the labels, either ground-truth labels $\mathbf{Y}$ or the temperature scaled (all activations are divided by a 'temperature' T to control how much to rely on the teacher's predictions) softmaxed probabilities from the large model $\boldsymbol{Y}^{(t)}$, and the temperature scaled softmaxed probabilities from the small model $\boldsymbol{Y}^{(s)}$ as input. λ_1 and λ_2 determine the relative importance of the ground truth loss $L_{ground-truth}$ and the distillation loss $L_{distillation}$. The idea is that this will help the small models get more insights from the subtle relations between the probabilities of different labels [16].

The article *Exploring Challenges of Deploying BERT-based NLP Models in Resource-Constrained Embedded Devices* [17] studied ways of finding the optimal balance between accuracy, energy consumption, inference time, and memory usage of BERT models. BERT (Bidirectional Encoder Representations from Transformers) models are essentially models that consist of only the encoder part of a transformer. Their function is to encode the meaning of a sequence and from this encoding make a classification or some other kind of prediction. A BERT model could possibly be used in my problem to encode the sequence and make some kind of classification of the sequence instead of generating the whole target sequence. From this classification some other model or hand-written code could be used to gather the message. The conclusions drawn from BERT models considering resource efficiency and accuracy could be generalizable to transformers, since they are compromised of the same parts: attention blocks, normalization and softmax layers, and feed-forward layers.

The study explores two ways of reducing the size of the model, pruning weights, and removing layers. The study provides a number of interesting insights and conclusions. The first one is that, depending on the task, the memory consumption can be up to five times higher than the size of the model, and the memory consumption, MC is not directly proportional to the model size, MS, but rather follows a formula that looks like $MC = aMS + b$, for some of the smaller models, the ground memory usage, b, made up two-thirds of the memory consumption. This explains why certain models consume more memory than expected, something later encountered in the study of the transformer model. The study also indicates that retaining the layers and pruning a larger percentage of the weights give better performance for the same model size, a finding worth to keep in mind when optimizing a transformer. Furthermore, the study found that pruning does not have a significant impact on energy consumption, which suggests that the choice of model type and the way the problem is formulated and solved is the most important when it comes to energy efficiency.

There seems to be much promise in pruning, since some of the studies discussed in the paper *Methods for Pruning Deep Neural Networks* [18] by Vadera and Ameen, have managed to remove well over 90% of the weights without affecting the accuracy. The authors highlight a paper from 2019 that has gotten much attention in the last years, called *The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks* [19], which formulates and shows through experiments that the lottery ticket hypothesis is valid. The hypothesis states that dense, randomly-initialized feed-forward networks contain sub-networks that when trained in isolation with the same initial weights have the same accuracy as the whole network. In their study they explore both one-shot pruning which removes $p\%$ of the weights at once, chosen by magnitude pruning, and then resets the weights to their original values and retrains the new network, and iterative pruning that during n cycles removes $p^{1/n}$ of the weights, retrains the network from initial values, and then choose new weights to prune based on magnitude. By iterative pruning, which they find more accurate than one-shot pruning, they manage to reduce the whole network to 3.6% of the original size retaining the same accuracy, and show that this smaller network

learns faster than the original large one, a very interesting finding for future studies in reducing the size of the transformer.

But a very important point that Vadera and Ameen [18] make is that these pruned sparse neural networks require specialised libraries or hardware to convert the sparsity into faster models, while pruning at a higher level, removing filters and channels, can more directly lead to improved performance. For this type of pruning they show evidence that earlier layers in a model are less sensitive to pruning and heavier pruning can be applied there. Another important aspect that improves pruning is fine tuning of the model after pruning, which is done by training the model after the pruning has been done. A promising technique for this is to use iterative pruning and fine-tune the model with just one or two epochs of training, before moving on to the next round of pruning to let the weights adjust to the changes in the model.

In a paper from 2020, Yu et al. [20] confirm that the lottery ticket hypothesis can be transferred to NLPs, and by iterative pruning and late rewinding (resetting weights not to their initial values but values they had some epochs later) they manage to reduce the size of transformer models to one third of the original size while maintaining the accuracy. Used right this could be very beneficial for the transformer model studied in this project.

Two papers that target especially computationally costly parts of the BERT model are *MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices* [21] and *SqueezeBERT: What can computer vision teach NLP about efficient neural networks?* [22]. In the first paper, linear-layers, referred to as bottlenecks, are added to reduce the width of the different blocks of the model. They scale down the original input dimension of 512 such that the hidden dimension of each layer is now 128 instead. They also implement residual connections between the blocks (feed-forward blocks or self-attention blocks) instead of just inside the blocks to get a faster information flow. Finally, as a third adjustment, they increase the number of feed-forward blocks in order to keep the ratio between the weights in the self-attention blocks and feed-forward blocks constant. Otherwise, the weights in the attention block would become a larger part of the total weights as the attention blocks get wider inputs than the feed-forward blocks as a consequence of the bottleneck layers [21]. The second paper builds on this architecture by replacing the position-wise fully-connected layers in the projection of the queries, keys and values, as well as in the feed-forward blocks, with 1D convolutional layers with a kernel size of 1, showing that they are numerically equivalent. Taking it one step further, they implement most of the convolutional layers as grouped convolutions, where the input matrix is split along its channel width into g different groups that each undergoes a separate convolution before concatenating the results. They call their model *SqueezeBERT*, and using student-teacher distillation they manage to get a model that is 4.3 times faster than the original, with almost the same accuracy [22]. These papers could serve as a guide to identifying and altering especially costly parts of the transformer architecture as well.

A paper that is interesting for multiple reasons is *Are Sixteen Heads Really Better than One?* by Michel, Levy, and Neubig [23]. It investigates the effects of pruning attention heads for both the original transformer architecture, with 6 layers of 16 heads each, and the standard BERT model, with 12 layers of 12 heads each. It can therefore provide insights in the validity of generalizing results from pruning BERT models to pruning transformer models.

The authors first investigate the effects of removing a single-head in the encoder part of the model. They find that the majority of the attention heads can be removed without significantly altering the accuracy if the rest of the model remains unchanged. They move on to removing

all attention heads but one, keeping only the best head, in one layer at a time of the model, concluding that for most layers a single head is enough for performance. However, some layers are more sensitive to pruning than others, and drastically deteriorate in performance when pruned to only one head. They also find that the importance of the attention heads is correlated between different data sets, showing that heads can be universally pruned before fine-tuning a model to its specific use. When comparing BERT and the transformer, they find that 40% of the heads can be removed in the BERT model without any noticeable negative effect, while the negative effects start to show already after pruning beyond 20% in the transformer. This would indicate a need for caution when generalizing the findings from BERT pruning to transformer pruning. But further studies of pruning heads in the transformer showed that the encoder-decoder attention (cross-attention) blocks are more sensitive to pruning while the encoder- and decoder-attention (self-attention) blocks are more robust and perform similarly to BERT models when pruned [23]. This highlights the importance of intelligent pruning, and also suggests where to apply the heaviest pruning in the transformer model.

3 Method

3.1 Data generation

To be able to train good and useful models, training data that is a good representation of the real data the models will encounter in use is essential. Unfortunately Saab did not have enough collected real-world data, especially not data that had been labelled. Therefore the generation of high quality, representative, and labelled data, has been a key part of the work load.

From the rules defining the structure of the codes, a code generator has been created in Python in which the user can define all aspects of a message that is going to be sent and get the corresponding time intervals.

The challenges with communication occurs when there is noise present in the code, therefore noise was simulated as well. Several types of noise were added to the original message. 10 percent of the intervals were shifted either up or down by one unit, representing uncertainty in the clocks at the sender and receiver. 1 percent of the pulses were removed, representing pulses getting lost on the way.

Finally, additional noise pulses were added from three different categories of noise, representing different challenges in signal decoding:

- The first category, defined as ordinary noise, was generated with an interval between each pulse randomly sampled from a normal distribution, with a mean value of 3000 tu and a standard deviation of 2500 tu.
- The second kind of noise was defined as structured noise. This noise was generated by repeating one of a small number of specific time intervals a number of times at random times in the code, in the end constituting around 1% of the final code.
- The final category of noise, defined as specific noise, came in the form of repeated time intervals from a wider range and from shorter intervals than the structured noise. This

noise was generated mostly to develop a simple filter for this specific kind of noise.

This meant that a message of length 3 000 000 tu contains around 1000 noise pulses, and a time interval of 1500 tu has a 50% risk of being split by a noise pulse. This resulted in a seriously distorted code with a noise frequency significantly higher than what the decoding can be expected to encounter. This was done with the goal of training a model with increased robustness for noisy codes.

As previously mentioned, an example of a message before and after noise has been added can be seen in Figure 1. The generated training data consisted of 30 000 correct messages and 15 000 faulty ones, while the test- and validation data consisted of 600 correct and 300 faulty messages.

In order to teach the models to recognize faulty code, faulty messages were also generated. For example, messages missing vital parts, or with parts sent in the wrong order, or messages where the time gap between the different repeated intervals was too long. Another reason for adding these faulty messages was to give the models the opportunity to classify code containing just noise as faulty code. Tests were also made where the hardest codes were made a larger part of the data. In order to be able to use the real data from Saab for evaluation of the models, labels for this data have also been generated.

3.2 Training models

3.2.1 Random Forest

The reason for using random forest was to implement an initial noise filter, that filtered most of the noise from the code before it was sent to the NLP model that made the real decoding of the message. The idea was that the NLP model could then focus on learning the patterns of the important parts of the messages instead of wasting energy at the noise. The NLP model would also get shorter sequences to process making both training and inference faster and more energy efficient.

The noise filter worked by classifying pulses as being either noise pulses or real pulses. The input features for each data point in the classifications were the time intervals to the preceding and subsequent pulses, illustrated in Table 1, here with eight features, four preceding and four subsequent time intervals.

Table 1: Description of how nine pulses (pulse zero at time 0 not displayed in the table) arriving at the given times, with eight corresponding intermediate time intervals in tu, are converted to eight features for pulse five (the one between time intervals 18 and 1327), and seven features for pulse four and six. Feature k is given by the time of arrival of pulse k minus the time of arrival of the current pulse. Pulse $k = 0$ is defined as the pulse four pulses before the current pulse when using eight features (five pulses before the current pulse using ten features, and so on). Zero-valued features appear for the start- and end pulses of an interval, where pulse k is not part of the range of incoming pulses.

Time of pulse arrivals:	269	807	1614	1632	2959	3252	4872	6492
Time intervals:	269	538	807	18	1327	293	1620	1620
Features for pulse 4:	0	-1614	-1345	-807	18	1345	1638	3258
Features for pulse 5:	-1632	-1363	-825	-18	1327	1620	3240	4860
Features for pulse 6:	-2690	-2152	-1345	-1327	293	1913	3533	0

The random forests were trained with the number of decision trees in the forest ranging from 1 to 10 and with max_depth of the trees ranging from 7 to 11. The goal was to find an optimal trade-off between the size of the forest, counted as the number of possible nodes if all internal nodes were branched, and the accuracy. The accuracy was calculated on validation data, generated with the same program as the training data. The entire filter was implemented as three successive filters. The second and third filter were trained and evaluated on the output from the previous filter. The three filters were:

- An initial filter which filtered out ordinary noise, trained with balanced class weights.
- A second filter once again filtering out ordinary noise, but with a ten times higher class weight for real codes. This filter was added to remove some additional noise pulses without removing real codes.
- A final filter, filtering out structured noise, trained with ordinary class weights.

A noise filter for the specific kind of noise mentioned earlier, carrying a higher risk of being classified as a real message, was also developed in the form of two successive filters. Considering the three different types of noise: ordinary, structured, and specific, the first filter was implemented in three versions: Either filtering out only ordinary noise, filtering out ordinary noise and specific noise, or filtering out all three types of noise. The second filter was implemented in two versions: Filtering out only specific noise, or both specific and structured noise. For these filters, all time intervals were considered positive, to allow for representing the data by unsigned integers. Evaluation of using the time intervals between successive pulses instead of summing the intervals before and after the pulse as features, was also carried out with the same set-up, this is equivalent to just using the second row of Table 1 as features for pulse 5.

3.2.2 Transformer

After the training data had been filtered through the best noise filters, and pulses classified as noise had been removed, the decoding of the messages in the filtered data remained. For the actual decoding task, a version of Tensorflow's transformer model [10] was trained, using the

same hyperparameters for the model as Tensorflow ($N = 4$, $d_{model} = 128$, $d_{ff} = 512$, and $num_{heads} = 8$), while experimenting with ways of applying the model to the problem. To avoid training the model on an unnecessarily large range of input tokens, intervals larger than twice the largest allowed interval in the original code (larger than around 5000), were assigned one of 18 large-number tokens, determined by the magnitude of the interval, before being sent as input to the transformer. Tests of some different formats of the target sequence, with the intent to make it easier for the transformer to translate the message, were done. More specifically, this meant formats with and without delimiter tokens around parts of the code that could be considered more closely related, and formats where the target sequence was either repeated the same number of times as the input sequence, or just repeated once regardless of the repetitions in the input sequence.

The accuracy score used to evaluate the models was the percentage of nonzero target tokens in the validation data the model got right. Tests of training the model on recognizing faulty messages, with or without the type of error, were carried out as well. The transformer model was also retrained several times after having altered the training data. The transformer was always trained from scratch for 20 epochs, where an epoch is defined as training the model on all samples once.

In the original transformer paper [3], the positional embedding was implemented as alternating sine- and cosine waves instead of the first half of the embedding dimension with sine waves and the second half with cosine waves as in the TensorFlow model [10]. Here the positional embedding was implemented both ways to evaluate the difference.

A smaller transformer model, with reduced model dimension (8), number of attention heads (2), and number of layers (2), made to fit on a certain micro controller, was tested. The small model had 65 000 parameters compared to the 8 300 000 parameters of the large model.

3.3 Implementation in C

As Branco, Ferreira, and Cabral point out in their article [24], programs exist that transfer trained models from one programming language to another, known as transpilers. But these transpilers are generic, made to create code that can run on any platform. They do not try to optimize the code and have other limitations [24]. Therefore, it is of value to study the process of exporting and translating a model from a high-level language to a low-level language by hand, to get insights in the process and evaluate the approach.

In this project, exporting of trained models from Python to C, has been studied, to enable implementation on the embedded platform. The forward-passes, making the predictions, of both the random forest and the transformer model were implemented from scratch.

3.3.1 Decision Tree and Random Forest

When exporting the noise filter and the specific noise filter from Python to C, five vectors were gathered from each tree:

- One vector stating the feature to evaluate.

- One vector containing the threshold condition for that feature.
- One class probability vector, twice the size of the other vectors, with a node occupying two entries. The first entry contains the number of data points from the training data, reaching that node, which were labelled as noise, and the second entry contains the number of data points that were labelled as something else.
- Two vectors giving the locations of the left and right child node of the current node.

The prediction of each tree was then calculated as the value of the first entry of the class probability vector divided by the sum of both entries. The predictions were then summed over all trees and divided by the number of trees to generate a final probability that the data point, or pulse, was noise and should be removed.

3.3.2 Transformer

The implementation of the transformer was done in two steps. The first step was to remodel all of the steps in the forward pass of the transformer. This was done by implementing the steps in Python, while rewriting functions that were not available in C. Most of the functions came down to different matrix multiplications and normalization procedures. For each part of the transformer, the input- and output matrices of the function from the original transformer were gathered. The new version of the function was then made to mimic the original output from the original input. Finally some minor changes were made to the new model to increase its prediction speed. The most significant one being to only run the encoder once for every sequence, and not make a new encoding before the prediction of every token in the output sequence, since the input sequence stayed the same.

The second step was to translate the implementation of the transformer from Python to C, using GNU Scientific Library (GSL) [25] for handling the matrices and their multiplications.

All parameters of the original transformer were collected in text files and copy-pasted into the C program, together with matrices containing input and output of every step in the transformer. The process of implementing functions was then repeated in C.

3.4 Evaluation and Testing

The random forests trained as noise filters were evaluated on a part of the training data that had been taken out of the training set and turned into a validation set. The aim was to correctly predict if a pulse was part of a message or just noise. All pulses from a type of noise the filter was not trained to identify were regarded as real code. The balanced accuracy was measured. This is an accuracy that is calculated one class at a time and then averaged over all classes, giving each class the same importance independently of the number of samples in each class. The average balanced accuracy of each forest- and tree size that was tested, from 10 random forests of each configuration, was plotted over the corresponding size, in an attempt to find configurations that were both accurate and small. The proportions of wrongly labelled noise pulses and wrongly labelled real codes of the best forest were sampled.

The transformer models were evaluated on a validation data set that had been generated in the same way as the training data, but with other random parameter values that gave other messages and noise pulses. Both the training data and the validation data had been filtered through the best noise filters before being given to the transformer. They were evaluated on accuracy in predicting each token in the target sequence, a task far more complicated than just labeling a pulse as noise or real.

For the final evaluation on the performance of the transformer versus the hand-written code and the hand-written code with additional filters, a point system was designed to reflect that certain tokens in the target sequence are more important than others. A prediction of the target sequence could earn between 1 and 0 points depending on which parts it got right. And if the program predicted a message when there was no valid message in the input sequence, the prediction got -1 points. Five different setups were tested: The transformer model with a preceding noise-filter, the hand-written code, the hand written code with a preceding noise-filter, the hand-written code with a preceding filter for specific noise, and the hand-written code with both filters added before the code.

The set-ups were tested on 600 valid and 300 faulty messages from self-generated data, and 57 valid and 8 faulty messages from the real data I got from Saab, and both the total score and the score for just the valid and the faulty messages were calculated and compared.

3.5 Optimization

In order to find ways of reducing the size and increasing the speed of the model, a study of the time needed for a prediction was made, with focus on finding any bottlenecks or especially slow parts of the model. The model was run in debugging mode in Visual Studio, and breakpoints introduced around the part of the code being measured. Visual Studio then measured the time it took to run the code from the first breakpoint to the second. A calculation of the required number of multiplications needed for predicting a sequence was also carried out to find out what influenced the prediction time the most, and if that time was reasonable. A measurement of the time needed for a 1000 by 2000 times a 2000 by 3000 matrix multiplication was done, using GSL, OpenBLAS [26], and a self-constructed naive implementation, using only for-loops, in order to investigate this way of speeding up the model.

Pruning, removing, weights based on magnitude, where the smallest x per cent of the parameters of a certain matrix in the transformer was set to zero, was tested. One-shot pruning was done, varying the percentage and number of layers, to find the maximum pruning possible without altering the first token predictions of the target sequences.

In order to reduce the size of the model, quantization was tested by changing the data types of the numbers involved in the calculations from double precision floating point numbers, occupying 64 bits each, to single precision floating point numbers, occupying 32 bits each, and observing the effects in the implemented transformer model in C.

4 Results

4.1 Training models

4.1.1 Random Forest

The results from one of the evaluations of accuracy versus forest size, for different number of trees in the forest, can be seen in Figure 4a). The accuracy is computed on validation data and the $size = trees \cdot 2^{max\_depth} - 1$.

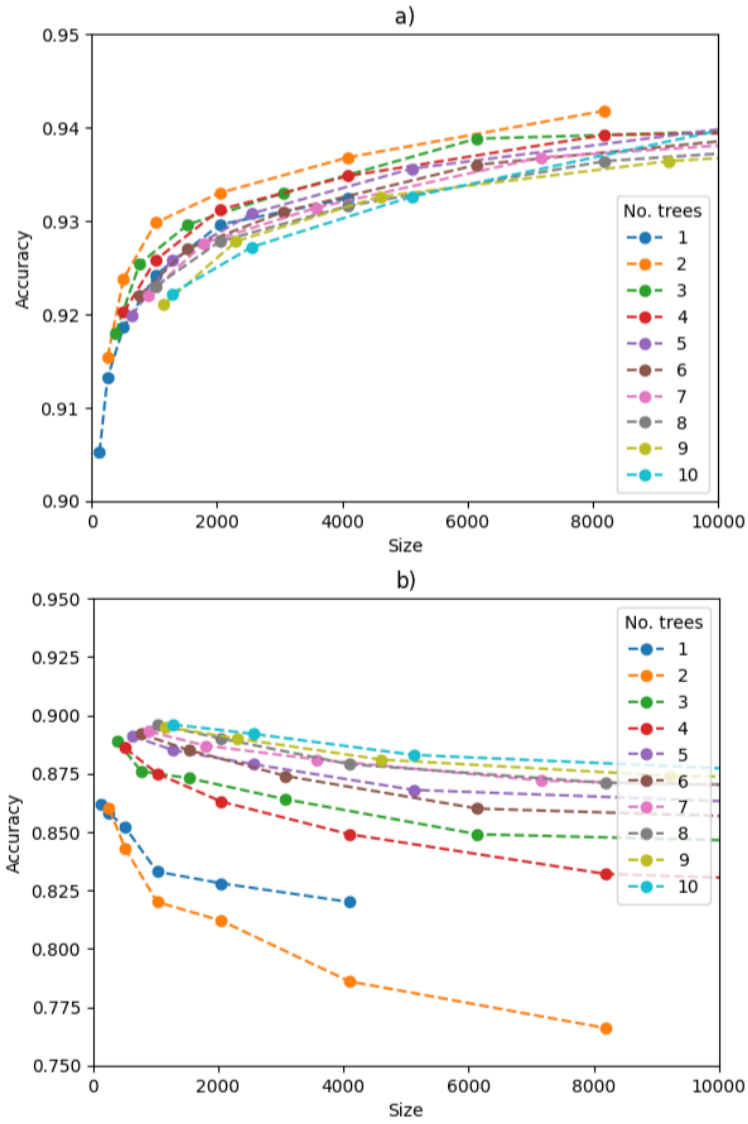

Figure 4: Plots of the balanced accuracy versus the maximum size of the forest, if all nodes were branched, for different number of trees (shown with different colours), and depths of the first noise filter .a) With models trained on the entire training data. b) With models trained on 0.1% of the training data.

We can see that with a lot of training data, as in Figure 4a), where the forests have been trained on 26 500 000 data points, the forests with few but deep trees are the best for the model-sizes

tested, possibly because of the higher maximum depth allowing for capturing more complexity with additional levels of nodes.

But it is interesting to see how the optimal choice of the number of trees and the maximum depth is affected by the size of the training data. In order to study this, the trees in Figure 4b) were trained using only 0.1% of the data. It seems that when there is little data available, as in Figure 4b), the forests with fewer but deeper trees seem to become over-fitted, and the forests with the largest amount of shallow trees perform the best. And adding more depth to the forests with the same number of trees also seems to increase the overfit, resulting in worse accuracy.

This is relevant to the work because of the connection to having representative training data. If the self-generated training data is only representative of a small part of the cases the model will encounter in inference, the same phenomenon can be expected to occur for the models trained on a small portion of the data. Therefore, if a developer suspects that the training data does not capture the entire data, it is better to use a higher number of shallow trees, to find broader, more general patterns. On the other hand, if the training data can be presumed to be of good quality, deeper trees with the ability to find more intricate patterns are beneficial.

The results from the test of the three versions of the first part of the specific noise filter are shown in Figure 5, using individual time intervals instead of accumulated intervals, which showed to be better.

From the plots it is evident that training the filter on just one type of noise at a time is beneficial for accuracy, since the random forests in Figure 5a) have the highest overall accuracy. The random forests seem to have detected patterns that are unique for the different kinds of noise, making it harder when they are all given the same label. This was also true for the evaluation of the second filter, letting the filter just classify the specific noise as noise gave higher accuracy. Thus, a developer working on classification with random forests should, if possible, try to evaluate breaking down the classification problem further, using successive classifications if necessary.

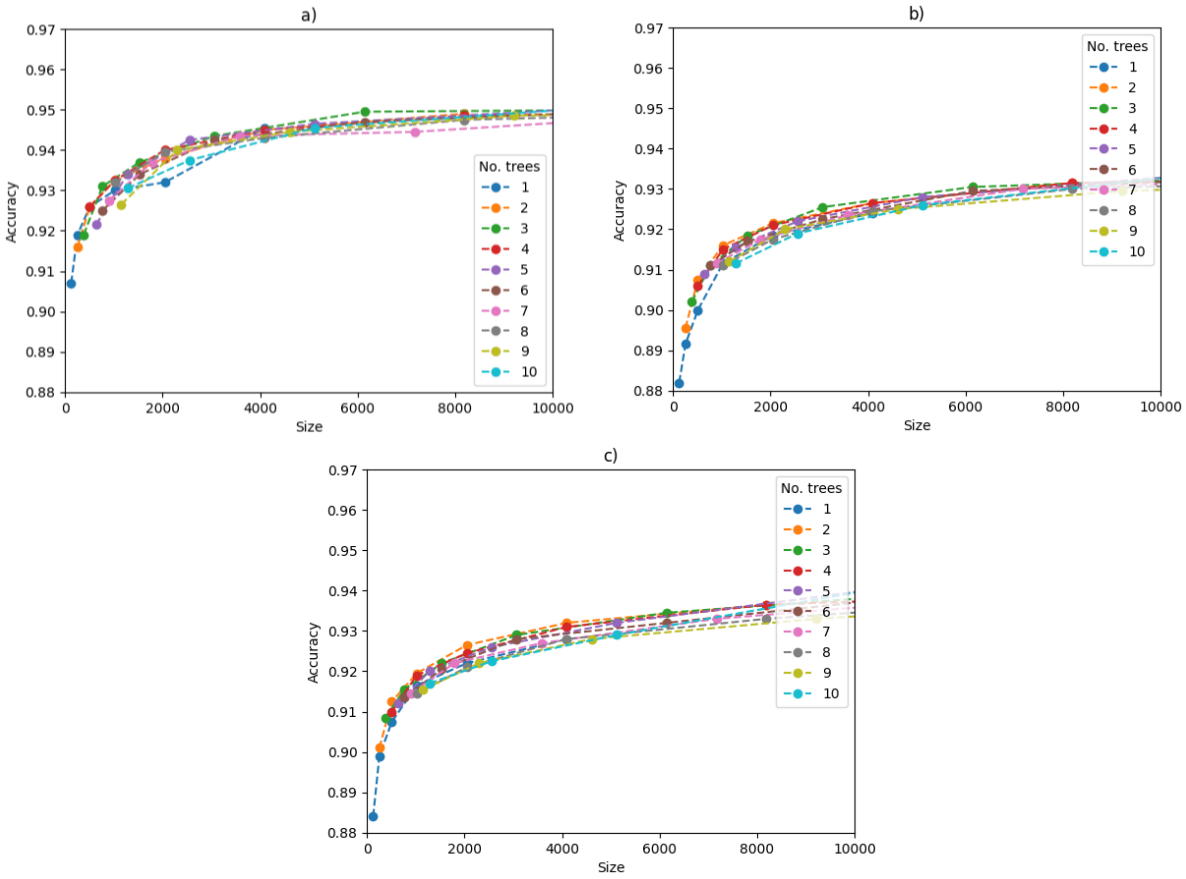

Figure 5: Plots of the balanced accuracy versus the maximum size of the forest, if all nodes were branched, for different number of trees (shown with different colours), and depths of the first specific noise filter. a) With only ordinary noise labelled as noise. b) With ordinary and specific noise labelled as noise. c) With all noise types labelled as noise.

4.1.2 Transformer

In Table 2 the accuracy and loss calculated from the validation data, during different parts of the training process can be found. For all but the first of the transformers in the table, the final target sequence consisted of a token describing the type of code sent, followed by blocks where each time interval appeared once, regardless of the number of successive times it was sent, surrounded by delimiter tokens, and in the end, an end of message token after the final block. This is an example of how the target sequence of an NLP model can be modified to not only do translation, but also classification (the first token) and grouping (the delimiter tokens), and thereby achieve better accuracy than ordinary translation. This enables the usage of the transformer as a signal decoding model, and points to the possibilities of modifying the transformer to handle other, novel, types of problems.

Table 2: The accuracy and cross entropy loss of the transformer model, calculated from the validation data, for different versions of the model and the training data, in order from oldest to newest. The highest accuracy and lowest loss are in bold.

Type of model/data:	Loss	Accuracy
With delimiters	0.196	0.942
Without delimiters	0.266	0.934
Faulty codes specifying error type	0.270	0.931
Faulty codes without error type	0.199	0.942
TensorFlow's positional embedding	0.194	0.946
The original positional embedding	**0.192**	0.943
Small transformer (50 epochs)	0.804	0.786
Removed faulty error codes	**0.192**	0.945
Removed additional faulty error codes	0.198	**0.947**

The tests of the different data formats showed that delimiters and multiple repetitions of the message gave better accuracy than without delimiters and with only one message. Shorter target-sequences, without delimiters and repetitions, would, on the other hand, result in shorter inference time. This trade-off between inference time and accuracy has to be considered by the developer.

When training the transformer on recognizing faulty messages, the accuracy dropped when the model tried to predict the type of error rather than just labeling all faulty messages with the same token. This indicated that it can be beneficial to let the transformer model only focus on a minimum required for solving the task. Trying to get it to learn more complicated patterns than that, seems to result in the model finding patterns where they do not exist.

To summarize the findings of the evaluation of different transformer models, the transformer does well on signal decoding by using a combination of classification and translation, which points to the possibility of using transformer models for a wider spectrum of problems than today. The study also seems to indicate that a less complex description of the problem is beneficial for the accuracy of the transformer.

The smaller model reached an accuracy of 0.79, after training for 50 epochs, compared to around 0.94 for the larger models trained on the same data (from "Faulty codes without error types", and forward in Table 2). But when using the small model in inference it generated sequences that were total gibberish, repeating certain tokens many times. This showed the need for a task specific evaluation protocol, which tested the performance in generating correct sequences rather than single tokens. This was why the message based scoring was set up at a later stage, in which a message was given 0 to 1 points based on which tokens it got right and -1 points were given if a message was detected in codes only consisting of noise.

4.2 Implementation in C

In the process of implementing the transformer model from scratch, several challenges with the approach of training a model in one language and then transferring the forward pass of the trained model to another language were discovered, and had to be overcome. The problems are

listed below:

• The translated implementation has to mimic the original model exactly, both when it comes to architecture and what happens in each calculation. Otherwise, the exported weights will not make sense, since they are trained on that exact architecture. This requires the developer to understand every single calculation in the model, which in this project led to many hours of studying the source code of the functions in the transformer.

• A related problem is that the representation or handling of numbers can differ between the languages. This should not affect the final inference if you look at the success that research has had with quantization, which alters the representations of numbers. But it could be a problem during translation of the model, when you want to check if the implementation of a certain part of the model yields numerically identical results, in order to ensure a correct translation. If the results differ, then the question arises if this is due to an error in the implementation or due to representational differences between the languages. In this work, an understanding of what errors could be expected in Python versus in C had to be developed before the implementation in C went smoothly.

• Higher level functions used in the original model, for example reshaping of matrices, might not be available in the target language. They must then be implemented from scratch in the new language, which requires even more understanding and time used for programming from the developer. This further prolonged the implementation in C in this project.

• The idea with this approach is to take advantage of the advanced programs for choosing, designing, and training deep learning models available in high-level programming languages, and then export the model to the low-level target language that allows it to be incorporated into embedded devices. The developer should therefore not have to study and understand the exact mechanics of how the model is trained. This saves time and resources, but restricts the changes the developer can do to the more complicated parts of the original architecture. In this work, it prevented a study of using a more computationally cheap normalization layer than a softmax layer in the attention blocks, since this would also have required implementing the training routines of these attention blocks.

• A similar restriction is that changes in the architecture can not be made solely in the translated model, but have to be made simultaneously in the original model, which should then be retrained and the new weights exported to the translated model. A process that can be costly for large models.

• Usually, many optimizations have been done in the code underlying the original model. Several operations in higher-level code are actually implemented using low-level, highly-optimized, near-machine code. These optimizations do not transfer with the translated model and the model can end up becoming slower after translation. This happened with the transformer model in this project which became around 10 times slower.

• The handling of the exported parameters also provides a challenge. To avoid wasting inference time reading files, the parameters should be implemented as a part of the model that is compiled together with it. The developer then has to find a clever way of creating files with exported parameters that have the right format to be read and incorporated into the model during compilation. Or the developer can choose to copy and paste the parameters from a .txt file into a source file. This can be quite time-consuming, and requires the developer to copy and paste all

parameters every time the original model is changed. Something that was made several times during the implementation phase of this project.

The implementation of the random forest was more straightforward and could be done without encountering any significant problems.

4.3 Evaluation and Testing

In Table 3 the accuracy scores and the false positives and negatives from the confusion matrices of the different random forests chosen for the noise filters can be found.

Table 3: The balanced accuracy (o=ordinary accuracy) and the proportion of wrongly classified noise and wrongly classified real code for the random forests that performed best for each filter type. N stands for noise, RC for real code, NF for noise filter, SF for specific noise filter, iv for individual time intervals, as opposed to cumulative time intervals. The first three filters were trained without specific noise in the training data, while the rest of the filters had this noise in their training data. The first three filters are the filters that filtered the data given to the transformer. The best values for the noise filters are coloured in red and the best values for the specific filters are coloured in blue.

Filter description	Trees	max_depth	Size	Accuracy	N class. as RC	RC class. as N
First NF	2	11	4095	0.933	0.103	0.0304
Second NF	5	10	5119	0.922 (o)	0.528	0.0111
Structured NF	2	10	2047	0.906 (o)	0.802	0.00419
NF v1	4	10	4095	0.943	0.0648	0.0500
NF v2	7	9	3583	0.907	0.0859	0.0982
NF v3	6	9	3071	0.912	0.0983	0.0767
SF v1	6	9	3071	0.908	0.159	0.0247
SF v2	6	9	3071	0.864	0.222	0.0497
NF (iv) v1	4	10	4095	0.940	0.0620	0.0570
NF (iv) v2	4	10	4095	0.926	0.0688	0.0783
NF (iv) v3	2	11	4095	0.933	0.0682	0.0664
SF (iv) v1	2	11	4095	0.953	0.0745	0.0200
SF (iv) v2	2	11	4095	0.919	0.122	0.0393

The results from the first noise filter show that the filter tends to more often wrongly classify noise than real codes even though the filter was trained with balanced accuracy, this might be because of the more diverse nature of the noise. The second noise filter and the structured noise filter both require less depth in the trees, and the structured noise filter is also smaller in size than the other two. Since they are successive filters, each filter gets less data to train on than the filter before. With less data, the benefit of larger, and eventually even more trees, seems to vanish.

When specific noise was present in the training data (NF v1 to SF (iv) v2), the filters show a better performance for all filters (except for NF v1) when using individual intervals instead of cumulative intervals. This could be because of how the subsequent intervals can be made irrelevant by the occurrence of a large interval, since the value of the feature will be above the

threshold, regardless of those later intervals, when using cumulative intervals. This is not the case when using individual intervals and therefore the features become more uncorrelated. The individual intervals also favours smaller but deeper forests, allowing for greater complexity.

To summarize, more data allows for larger forests, and it seems to be beneficial to have uncorrelated features, as in the case of the individual time intervals, when training random forests. As previously mentioned, the versions (v1) that focus on just filtering one type of noise are also superior.

The results from the tests of message decoding on self-generated data are shown in Table 4. And the results from the tests of real data are shown in Table 5.

Table 4: The average message accuracy points for different setups, where the accuracy depends on how much and which parts of the final message the prediction got right, a prediction could earn between 1 and 0 points and a predicted message where there was none gave -1 points. The hand-written code is referred to as Current, SF stands for specific filter and NF for noise filter. Here, the setups were evaluated on self-generated data. The values are presented relative to the current code and the best values are in bold.

Configuration	Total message acc.	Real message acc.	Erroneous message acc.
Transformer	**1.22**	**1.28**	1.04
Current code	1.00	1.00	1.00
Current + SF	0.78	0.54	**1.35**
Current + NF + SF:	0.76	0.51	1.33
Current +NF	0.77	0.53	1.33

Table 5: The average message accuracy points for different setups, where the accuracy depends on how much and which parts of the final message the prediction got right, a prediction could earn between 1 and 0 points and a predicted message where there was none gave -1 points. The hand-written code is referred to as Current, SF stands for specific filter and NF for noise filter. Here, the setups were evaluated on data from Saab. The values are presented relative to the current code and the best values are in bold.

Configuration	Total message acc.	Real message acc.	Erroneous message acc.
Transformer	0.38	0.035	1.00
Current code	**1.00**	**1.00**	1.00
Current + SF	**1.00**	**1.00**	1.00
Current + NF + SF:	0.78	0.66	1.00
Current +NF	0.78	0.66	1.00

To visualise the results they were also plotted in Figure 6a), which contain the results from the self-generated messages, and in Figure 6b), which contain the messages from Saab. The final evaluation showed that the transformer model, with an initial noise-filter, performed better on the self-generated data (Figure 6a)), than the current code, both for valid and faulty messages. When adding different noise filters, the program got better at discarding faulty messages, but also discarded a significant part of the valid messages. The best model with an added filter was the current code with a preceding filter for specific noise.

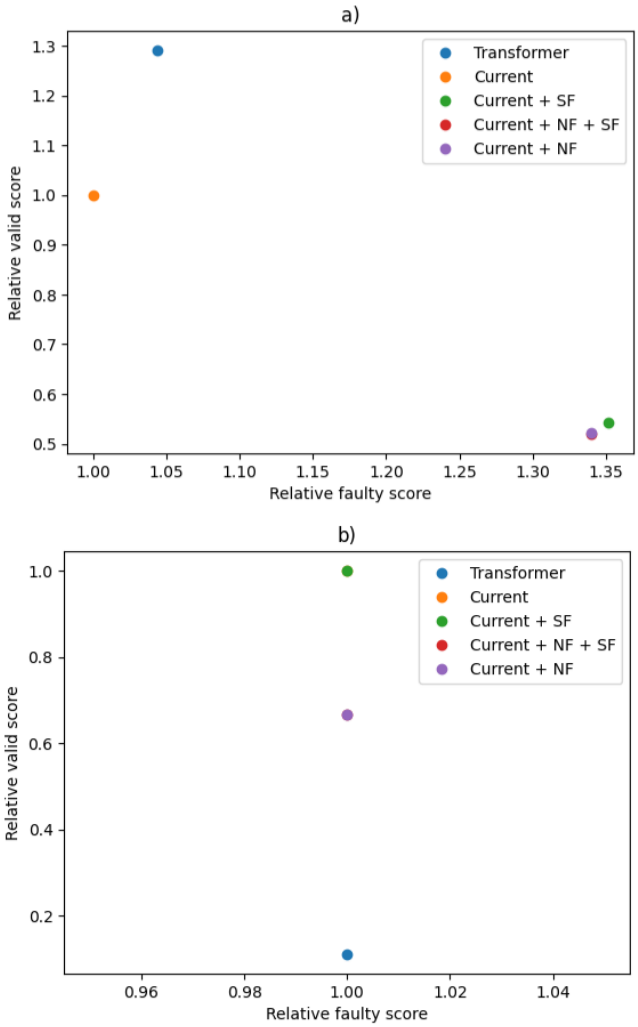

Figure 6: Plots of the average scores of the valid and faulty messages for the final evaluation of five different setups, relative to the hand-written code. The hand-written code is referred to as Current, SF stands for specific filter and NF for noise filter. a) Evaluated on self-generated messages. b) Evaluated on Saab messages. The specific filter did not alter the values, therefore the datapoints overlap.

The messages from Saab turned out to be more difficult to predict. No model out-performed the current code on the valid messages. The faulty messages were, however, easier to discard, as all models managed to accurately classify them all. The current code performed the best, and the specific filter had no impact on the accuracy when it was applied. The transformer model performed badly on these messages, classifying most of the valid messages as faulty.

The conclusion becomes that the transformer model performs well on data that is similar to the data it has been trained on, with significantly more noise than the data from Saab. But that it is also overfitted on the training data. It would therefore be beneficial to also include data without noise in the self-generated training data, perhaps it would then be possible to have the transformer model perform better at Saab's data as well. The filters make the model perform better on recognizing faulty messages, but in their current configuration seem to filter out too much of the real codes as well. Perhaps this was a major part of the problem with the transformer, which had an initial noise filter as part of the model.

4.4 Optimization

While the random-forest was successfully implemented on the micro controller, the transformer model required 20 times more working memory than the micro controller had capacity for, and also had an inference time 20 times longer, on a CPU, than what could be accepted on the target platform. Since the model was supposed to run on a micro controller with significantly lower computational capacity than the CPU, the inference time was the most severe problem.

The results from the analysis of the different parts of the inference time and the corresponding number of multiplications are shown in Table 6.

Table 6: Analysis of the time and the major part of the number of multiplications required for each part of the transformer for an input-sequence length of 150 and target-sequence length of 30. $head$ is the number of attention heads, enc is the encoder sequence length, dec is the decoder sequence length, seq is a general sequence length, and (enc) stands for an analysis done only in the encoder.

Part of the model/process	Times [ms]	No mult. -expression	No mult. -result
Translation			
Total translation	20 700	$16 \cdot heads \cdot enc \cdot d_{model}^2 + 16 \cdot heads \cdot enc^2 \cdot d_{model} + 8 \cdot enc \cdot d_{model} \cdot d_{ff} + 1.5 \cdot dec \cdot 8 \cdot heads \cdot enc \cdot d_{model}^2$	$7.8 \cdot 10^9$
Encoding	1 970	$16 \cdot heads \cdot enc \cdot d_{model}^2 + 16 \cdot heads \cdot enc^2 \cdot d_{model} + 8 \cdot enc \cdot d_{model} \cdot d_{ff}$	$7.6 \cdot 10^8$
Decoding	439-832	$8 \cdot heads \cdot enc \cdot d_{model}^2 + 24 \cdot heads \cdot dec \cdot d_{model}^2 + 8 \cdot dec \cdot d_{model} \cdot d_{ff}$	$1.6 - 2.7 \cdot 10^8$
Encoder layer			
Self attention	342	$4 \cdot heads \cdot enc \cdot d_{model}^2 + 2 \cdot heads \cdot enc^2 \cdot d_{model}$	$1.2 \cdot 10^8$
Layer normalization	3	$4 \cdot enc \cdot d_{model}$	$7.7 \cdot 10^4$
Feed Forward layer	29	$2 \cdot enc \cdot d_{model} \cdot d_{ff}$	$2.0 \cdot 10^7$
Decoder layer			
Causal attention	3-32	$4 \cdot heads \cdot dec \cdot d_{model}^2$	$0.52 - 16 \cdot 10^6$
Cross attention	99-120	$2 \cdot heads \cdot enc \cdot d_{model}^2 + 2 \cdot heads \cdot dec \cdot d_{model}^2$	$3.9 - 4.7 \cdot 10^7$
2 feed forward layers	1-16	$4 \cdot dec \cdot d_{model} \cdot d_{ff}$	$0.26 - 7.9 \cdot 10^6$
Feed forward			
First linear layer	24 (enc)	$seq \cdot d_{model} \cdot d_{ff}$	$9.8 \cdot 10^6$ (enc)
ReLU	2 (enc)	$2.5 \cdot seq \cdot d_{ff}$	$1.9 \cdot 10^5$ (enc)
Second linear layer	3 (enc)	$seq \cdot d_{model} \cdot d_{ff}$	$9.8 \cdot 10^6$ (enc)
Layer normalization	4 (enc)	$4 \cdot seq \cdot d_{model}$	$7.7 \cdot 10^4$ (enc)
Self attention			
Projection of Q and K	100	$2 \cdot heads \cdot enc \cdot d_{model}^2$	$3.9 \cdot 10^7$
Dot product attention	57	$heads \cdot enc^2 \cdot d_{model}$	$2.3 \cdot 10^7$
Softmax	11	$3 \cdot heads \cdot enc^2$	$5.4 \cdot 10^5$
Projection of V	50	$heads \cdot enc \cdot d_{model}^2$	$2.0 \cdot 10^7$
Weighted sum	62	$heads \cdot enc^2 \cdot d_{model}$	$2.3 \cdot 10^7$
Final linear layer	54	$heads \cdot enc \cdot d_{model}^2$	$2.0 \cdot 10^7$
Cross attention			
Projection of Q	1-10	$heads \cdot dec \cdot d_{model}^2$	$0.13 - 3.9 \cdot 10^6$
Projection of K	49	$heads \cdot enc \cdot d_{model}^2$	$2.0 \cdot 10^7$
Projection of V	48	$heads \cdot enc \cdot d_{model}^2$	$2.0 \cdot 10^7$
Final linear layer	1-10	$heads \cdot dec \cdot d_{model}^2$	$0.13 - 3.9 \cdot 10^6$
Causal attention			
Projection of Q	1-10	$heads \cdot dec \cdot d_{model}^2$	$0.13 - 3.9 \cdot 10^6$
Projection of K	1-10	$heads \cdot dec \cdot d_{model}^2$	$0.13 - 3.9 \cdot 10^6$
Projection of V	1-10	$heads \cdot dec \cdot d_{model}^2$	$0.13 - 3.9 \cdot 10^6$
Final linear layer	1-10	$heads \cdot dec \cdot d_{model}^2$	$0.13 - 3.9 \cdot 10^6$

By looking at the total translation, we find that the number of multiplications can be approximated by:

$$\begin{aligned} &16 \cdot heads \cdot enc \cdot d_{model}^2 + 16 \cdot heads \cdot enc^2 \cdot d_{model} \\ &+8 \cdot enc \cdot d_{model} \cdot d_{ff} + 1.5 \cdot dec \cdot 8 \cdot heads \cdot enc \cdot d_{model}^2 \\ &= 2.6 \cdot 10^6 \cdot enc + 1.6 \cdot 10^4 \cdot enc^2 + 1.6 \cdot 10^6 \cdot enc \cdot dec \end{aligned}$$

.

Using a typical encoder sequence length of 150 and a decoder sequence length of 30 the addends become $3.2 \cdot 10^8 + 3.7 \cdot 10^8 + 7.5 \cdot 10^9$. Thus the third term dominates and the number of multiplications for a translation, and therefore the inference time, becomes proportional to the product of the decoder and encoder sequence lengths.

The results revealed that it is the matrix-matrix multiplications that were the most time-consuming. The softmax layers took more than ten times longer per operation, due to the exponent operations, but were such a small part of the entire model that they did not have a significant impact on the total inference time. Another interesting finding of the analysis was that one specific matrix-matrix multiplication was around ten times faster than what could be expected from the number of multiplications. This was because of the preceding ReLU layer, which set a large number of the activations to zero.

The total number of multiplications needed for inference of a typical sequence was found to be around $7 \cdot 10^9$, which, if the full potential of the CPU of 700 GFLOPS (billion (singel-precision) floating point operations per second) [27], should take around 10 ms. But the program took 25 seconds to run, more than 2000 times longer. A comparison of the time required for a large matrix-matrix multiplication, using GSL, OpenBLAS, and a naive implementation, was done. No significant differences in the required time were found, although OpenBLAS has been reported to be more than 150 times faster [28] than a naive implementation. This could either be because the compiler in Visual Studio has already made enough optimizations of the naive code, or because OpenBLAS does not manage to achieve any optimization. However, this optimization would not transfer to the micro processor anyway, so probably the issue of speed would occur again when implementing the model on the embedded platform, and have to be solved in another way.

The maximal pruning possible before altering the predicted first tokens in the target sequences, signifying code type, was the removal of 39% of the weights, in the projection layers of the queries, keys and values of the self-attention in the encoder and decoder.

The quantization tests resulted in an almost halved memory consumption and a 20% reduction in inference time, without affecting the predicted sequences.

The point of showing the results of the matrix-multiplication-speed analysis, and the pruning and quantization results, is to suggest that the transformer model could very well possibly be implemented on an embedded platform, with better pruning, further quantization, more optimized software, and possibly using some of the other methods discussed in the literature study. Or rather, that more studies of possible improvements are necessary before discarding the transformer model in embedded environments.

5 Discussion

The work during this project has to a high degree been about solving the particular problem with the decoding of Saab's codes. As a consequence, some of the more scientific aspects of the experiments have not been fully pursued. Many of the minor studies have been either just a proof of concept to show what could be done or a small investigation into a topic to get a feeling of if this is a path worth investing time in.

The limited time for the project has led to a restriction of the studies of deep learning models to just two major types, one very simple, and one rather large and complicated. It is possible that other conclusions regarding what was possible to implement in an embedded environment would have been reached if more models had been tried. Especially models with a complexity in the middle of those two tried. Maybe a series of classificational neural networks, perhaps with convolutions, would have been able to solve the decoding problem as well as the transformer did. Or a BERT model just using the encoder part of the transformer followed by some clever hand-written extraction algorithm could have been enough to solve the problem. These are all things that a further study could investigate.

There is also a general problem with using test data that have been artificially generated with the same algorithms that generate the training data. Since it is of huge importance for the performance of the models to have training data representative of the conditions where the models will be deployed, the training data will probably change during the development of the models, when additional cases and inaccuracies are discovered. When this happens, the test data is also changed, making it hard to tell if the changes made the models better or worse. Some insight could probably be gained by keeping old versions of the test data, but the inaccuracy of the older data then has to be taken into consideration when using it for evaluation. At least when changes are made only to enhance the training of the model, for example by increasing the occurrence of especially tricky cases, the performance should be evaluated on older test data. But this was not done in an accurate way in this work. It can also be questioned how representative the 65 real messages from Saab are for the entire data the models will encounter in usage.

Looking back at the test of the transformer with delimiters, it is possible that the delimiters might be easier to predict than the rest of the tokens, therefore possibly increasing the accuracy without the model being better at predicting the actual tokens. A related discussion is whether the drop in accuracy found when letting the transformer predict error type is an accuracy drop for all tokens and not just the token representing the type of error. There could be a point in investigating both these questions further to get a better model.

Even though the transformer showed to be too large and too slow for deployment, a conventional understanding in the deep learning community is that a large number of connections is necessary during training, but once the network has been trained, a large portion of the parameters are redundant [29]. This corresponds to the findings that a large model trained on a large data set can help a much smaller model reach the same accuracy. This opens up the potential of reducing the size of the transformer and perhaps be able to deploy it on the embedded platform. Some of the studies on pruning and distillation have been able to show a four times reduction in size, and combining this with a quantization of both activations and parameters using 8-bit fixed point numbers, would result in another four times reduction, for a total of a 16 times reduction in memory consumption. Using a different way of computing the biases, that doesn't

require repeating the weights a number of times equal to the maximum sequence lengths, could possibly ultimately achieve the 20 times reduction needed. This would also make the model faster, which could be further enhanced by changing the representation of the target sequence, reducing its length significantly. Combining this, it might be possible to run the model on the intended micro processor. In the future, novel types of models could also be invented that turn out to be more resource efficient, enabling further deployment of complex models in embedded environments.

6 Conclusion

This work has shown that a transformer model can be successfully applied to decoding and recovering corrupt messages, by using a combination of classification and translation. But the standard models have shown to be too large and too slow for deployment on embedded devices in their current form. A reduction in size of about 20 times and a speed-up of at least 20 times would be needed to make deployment on the certain microprocessor possible.

The challenges of exporting a deep learning model are numerous as I have discovered during this work, from the need to exactly replicate all steps in the original model to the loss of functions and optimization. This should definitely be taken into consideration when choosing training in one language and inference in another as an approach to embedded machine learning.

When it comes to choosing the right kind of random forest for an embedded platform, an evaluation of the quality of the training data should be done. Deep forests are only good if the training data is sufficiently representative of the entire data, otherwise they tend to overfit. It is also better to let the random forests focus on classifying one thing at a time, breaking down the problem.

I have also touched upon the challenges of using self-generated training data when the labelled training data is scarce or kept secret by the customer. The generation of accurate data, representative of real-world conditions, becomes more important for the performance of the model than fine-tuning hyperparameters. And it becomes especially challenging when the generation of training data becomes an integrated part of the model development, in such a way that the validation data keep changing during the work.

The final evaluation of the transformer model with an initial noise filter compared to different combinations of random-forest filters and the hand-written code showed that the transformer-model had a potential for solving the problem better than the previous code, but right now it was over-trained on codes with a lot of noise. If the problem was just filtering out unwanted messages this could be done by a much simpler random-forest model that is easily implemented on an embedded platform, but right now these are also over-trained and filters out to much.

Further studies on accuracy versus energy consumption comparing more different machine-learning models on embedded platforms would be beneficial for choosing what to focus on when developing machine learning for the increasing number of products that are part of the, so-called, Internet of Things.

7 References

[1] OpenAI, "ChatGPT (Mar 23 version)," 2023. https://chat.openai.com/. Accessed: 2023-05-03.

[2] F. Duarte, "Number of ChatGPT Users (2023)," 2023. https://explodingtopics.com/blog/chatgpt-users. Accessed: 2023-05-03.

[3] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Llion, J., Gomez, A., Kaiser, L., and Polosukhin, I., "Attention Is All You Need," *Proceedings of the 30th Annual Conference on Neural Information Processing Systems*, pp. 5998–6008, 2017.

[4] Vanian, J. and Leswing, K., "ChatGPT and generative AI are booming, but the costs can be extraordinary," 2023. https://www.cnbc.com/2023/03/13/chatgpt-and-generative-ai-are-booming-but-at-a-very-expensive-price.html. Accessed: 2023-05-03.

[5] H. of Data Science, "Decision Tree and Random Forest Algorithms: Decision Drivers," 2021. https://www.historyofdatascience.com/decision-tree-and-random-forest-algorithms-decision-drivers/. Accessed: 2023-04-27.

[6] IBM, "What is a Decision Tree?," 2023. https://www.ibm.com/topics/decision-trees. Accessed: 2023-04-27.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[8] Scikit-learn, "sklearn.ensemble.RandomForestClassifier," 2023. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. Accessed: 2023-04-28.

[9] Scikit-learn, "1.10. Decision Trees," 2023. https://scikit-learn.org/stable/modules/tree.html. Accessed: 2023-04-28.

[10] Abadi, M., Agarwal, A., and Barham, P. et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. https://www.tensorflow.org/. Accessed: 2023-05-12.

[11] He, K., Zhang, X., Ren, S., and Sun, J., "Deep residual learning for image recognition.," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

[12] Ba, J. L., Kiros J. R., and Hinton, G. E., "Layer Normalization," *arXiv preprint*, vol. 1607.06450, 2016.

[13] Srivastava, N., Hinton, G. E., Krizhevsky, A, Sutskever, I, and Salakhutdinov, R, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of Machine Learning Research*, vol. 15(1), pp. 1929–1958, 2014.

[14] Kingma, D. and Ba, J., "Adam: A method for stochastic optimization.," *ICLR (3rd International Conference in Learning Representations)*, 2015.

[15] Koech, K. E., "Cross-Entropy Loss Function," 2020. https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e. Accessed: 2023-05-11.

[16] G. Menghani, "Efficient Deep Learning: A Survey on Making Deep Learning Models Smaller, Faster, and Better," *ACM Computing Surveys*, vol. 55, no. 12, 2023.

[17] Souvika Sarkar, Mohammad Fakhruddin Babar, Md Mahadi Hassan, Monowar Hasan, and Shubhra Kanti Karmaker Santu, "Exploring Challenges of Deploying BERT-based NLP Models in Resource-Constrained Embedded Devices," *arXiv preprint*, vol. 2304.11520, 2023.

[18] Sunil Vadera and Salem Ameen, "Methods for Pruning Deep Neural Networks," *IEEEAccess*, vol. 10, pp. 63280–63300, 2022.

[19] J. Frankle and M. Carbin, "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks," *International Conference on Learning Representations*, 2019.

[20] H. Yu, S. Edunov, Y. Tian, and A. S. Morcos, "Playing the lottery with rewards and multiple languages: lottery tickets in RL and NLP," *International Conference on Learning Representations*, 2020.

[21] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, and D. Zhou, "MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices," *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL*, pp. 2158–2170, 2020.

[22] F. Iandola, A. Shaw, R. Krishna, and K. Keutzer, "SqueezeBERT: What can computer vision teach NLP about efficient neural networks?," *Proceedings of SustaiNLP: Workshop on Simple and Efficient Natural Language Processing*, pp. 124–135, 2020.

[23] P. Michel, O. Levy, and G. Neubig, "Are Sixteen Heads Really Better than One?," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[24] S. Branco, A. G. Ferreira, and J. Cabral, "Machine Learning in Resource-Scarce Embedded Systems, FPGAs, and End-Devices: A Survey," *Electronics*, vol. 8, no. 11, 2019.

[25] Galassi, M., "GNU Scientific Library Reference Manual (3rd Ed.)," 2009. http://www.gnu.org/software/gsl. Accessed: 2023-05-15.

[26] Xianyi Z., Kroeker M., Saar W., Qian W., Chothia Z., Shaohu C., and Wen L., "OpenBLAS," 2023. http://www.openblas.net/ Accessed: 2023-05-15.

[27] cpu monkey, "Intel Core i9-11950H Benchmark, Test and specs," 2023. https://www.cpu-monkey.com/en/cpu-intel_core_i9_11950h. Accessed: 2023-05-18.

[28] A. Sicard-Ramíres, "Intel Math Kernel Library Matrix Multiplication," 2018. https://www.eafit.edu.co/centros/apolo/Documents/Andres_Sicard_mk_slides.pdf Accessed: 2023-05-02.

[29] Sun, S., Cheng, Y., Gan, Z., and Liu, J., "Patient Knowledge Distillation for BERT Model Compression," *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 4323–4332, 2019.