Gottfried Wilhelm Leibniz Universität Hannover
Institut für Informationsverarbeitung

Project Report

An Empirical Analysis of Fine-Tuning Large Language Models for Predictive Maintenance

Anh Khoa Pham

Matrikelnr. 10022723

Betreuer:	M. Sc. Quy Le Xuan
Erstprüfer:	Prof. Dr.-Ing. J. Ostermann
Zweitprüfer:	Prof. Dr.-Ing. B. Rosenhahn

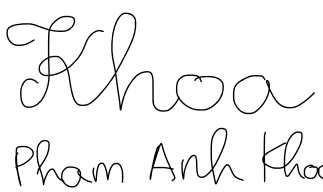
Hannover, Mai 2025

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie die aus fremden Quellen indirekt oder direkt übernommenen Inhalte und Gedanken als solche kenntlich gemacht habe. Dies schließt die Verwendung von elektronischen Medien sowie text- oder andere inhaltsgenerierenden IT-Werkzeuge wie ChatGPT oder GitHub Copilot ein. Ich versichere, dass alle abgegebenen Fassungen der Arbeit, im Besonderen gedruckte und elektronische Ausführungen, vollständig übereinstimmen und die Arbeit nicht zum Erwerb eines anderen Leistungsnachweises eingereicht wurde. Ich stimme der Verwertung meiner Arbeit durch das Institut für Informationsverarbeitung in den folgenden Punkten zu:

- Aufnahme der gedruckten und der elektronischen Fassung der Arbeit in die Institutsbibliothek,
- Übermittlung meiner Arbeit auch an externe Dienste zur Plagiatsprüfung,
- Vervielfältigung der gesamten Arbeit oder von Auszügen für Lehrzwecke und
- Wiedergabe der Arbeit durch Bild- und Tonträger.

Hannover, den 29. Mai 2025



Contents

Abstract

Traditional approaches to fault diagnosis in complex systems have relied on rule-based expert systems and manual inspections. These methods are often time-consuming, prone to human error, and require extensive domain-specific expertise. With the advent of machine learning, predictive maintenance has seen significant advancements, primarily through models trained on structured tabular data. However, such models typically function as "black boxes": they achieve high performance but lack transparency and adaptability to unseen failure scenarios.

This thesis explores a novel approach that leverages large language models (LLMs), both pre-trained and fine-tuned, for fault detection and predictive maintenance using tabular data. We investigate the potential of LLMs not only to identify faults and estimate time-to-failure, but also to generate human-readable explanations of their predictions. This interpretability addresses a key limitation of conventional models, which often fail to provide insight into their decision-making processes.

Experimental results demonstrate that fine-tuned LLMs can improve both diagnostic accuracy and interpretability. These findings suggest a promising direction for building more reliable, explainable, and trustworthy AI systems for industrial maintenance applications.

1 Introduction

In modern factories, machines must be reliable and work well to keep production smooth and meet market needs. Maintenance is very important to keep machines safe, efficient, and long-lasting.

Maintenance means taking care of machines during their whole life to make sure they work properly or to fix them when they don't [1]. Before machine learning, maintenance was done manually. There were two common ways:

- Run-to-Failure Management: Maintenance happens only after a machine breaks. This is simple but expensive and risky. A sudden breakdown can stop production and be dangerous for workers [2].
- Preventive Maintenance Management: Maintenance is planned based on time or how much a machine is used. This is better, but it can waste money if done too early or cause problems if done too late, especially if machines are used differently in different places [2].

Without regular maintenance, machines can break down unexpectedly. This causes delays, more downtime, and safety risks for workers. In industries like car manufacturing, electronics, or medicine, even a short stop can lead to big money losses and lower product quality.

Well-maintained machines use less energy, work better, and make fewer faulty products. Preventive and predictive maintenance find problems early, so they can be fixed before machines break. This makes machines last longer and saves money.

In Industry 4.0, technology is replacing old methods. Machine learning helps because collecting and storing data is now easier. This makes it possible to predict problems, find faults, and estimate how long machines will last [3].

This thesis studies how large language models (LLMs), both pre-trained and adjusted, can help find faults in complex systems using table data. LLMs are famous for working with text, but recent studies show they can also work with structured data. This research looks at how LLMs can improve fault detection, make decisions easier to understand, and provide better maintenance solutions for factories [4].

2 Theoretical Background

In this chapter, we will introduce deep learning methods that have been applied in the fields of prediction, fault prediction, and estimating the lifespan of machinery. At the same time, we will also explain the basics of how LLM works.

2.1 Deep Learning

DL algorithms often focus on training the layers of deep neural networks, which are inspired by the intelligent human brain and can process information using tightly connected cells [5].

2.2 Artificial Neural Network

Figure 2.1 below is a simple artificial neuron model consisting of only linear layers. Simply put, our input x will go into the Input layer, and the model will process the input in the Hidden layers and will return the output y in the Output layer. For a linear layer, the relationship between the input and the output is expressed by Equation 2.1.

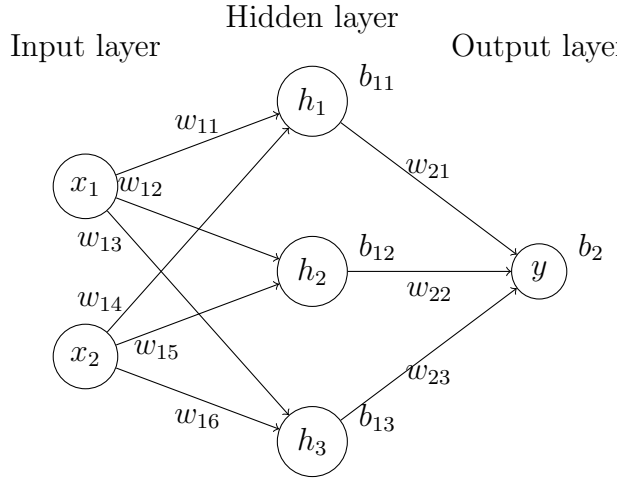


Figure 2.1: Simple neural network with fully connected layers

$$y = xw^T + b \tag{2.1}$$

At the end, weights w and biases b are the parameters that will be updated during the training process. In other words, we aim to obtain the parameters w and b from initially random values in such a way that the predicted values $\hat{y}$ are as close as possible to the ground truth y.

2.2.1 Activation Function

One of the important characteristics of artificial deep neural networks is their ability to represent nonlinear computations. This is made possible through activation functions that are added between the layers of the model. Commonly used nonlinear functions in neural networks include ReLU, Tanh, Sigmoid, Leaky ReLU, etc., as illustrated in Figure 2.2.

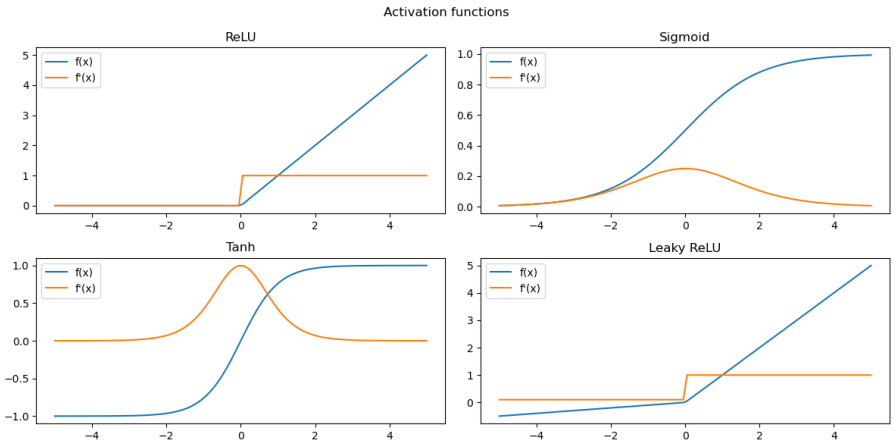


Figure 2.2: Activation functions

$$ReLU(x) = \max(0, x) \tag{2.2}$$

$$Leaky\_ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases} \tag{2.3}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.5}$$

The ReLU function in Equation 2.2 is the most commonly used due to its computational efficiency and the relatively simple process of computing gradients. However, it is not without its weaknesses. One notable issue is the dying ReLU problem, where all negative input values result in an output of zero [6]. This leads to zero gradients and prevents the model's parameters from updating. A common solution to this problem is to use

the Leaky ReLU function, as shown in Equation 2.3. In this case, if the input value is negative, it is multiplied by a small value α, where α is a hyperparameter [7].

The Sigmoid function will constrain the output within a range from 0 to 1. It is not only a nonlinear function but also commonly used in binary classification tasks. Meanwhile, the Tanh function constrains the output within a range from -1 to 1.

2.2.2 Loss Function

To measure the difference between the predictions and the ground truth, and to evaluate the performance of the model, we use a loss function

For regression tasks, we usually use loss functions like Mean Absolute Error (MAE), Mean Squared Error (MSE), and Root Mean Squared Error (RMSE) [8]. Their formulas are shown in Equations 2.7, 2.6, and 2.8, where N is the number of samples, y_i and $\hat{y}_i$ are true and predicted values, respectively.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{2.6}$$

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{2.7}$$

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2} \tag{2.8}$$

For classification tasks, the most commonly used loss functions are Binary Cross Entropy (BCE) when there are only two labels, and Cross Entropy Loss for multi-class problems [9]. Their formulas are shown in Equations 2.9 and 2.10, where N is the number of samples, y_i and $\hat{y}_i$ are true and predicted values and C is the number of the classes, respectively.

$$BCE = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i) \tag{2.9}$$

$$CE = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij}) \tag{2.10}$$

2.2.3 Stochastic Gradient Descent and Backpropagation

The loss function not only serves to evaluate the performance of a model but also plays a crucial role in updating the weights to train the model so that the output as predictions get as close as possible to the true values. In other words, we want to find the parameters

such as weights and biases that minimize the loss function. Stochastic Gradient Descent (SGD) is one of the fundamental algorithms used to train models [10], and it is expressed in Equation 2.11. We first calculate the gradient of the loss function with respect to the weight, then multiply it by the learning rate, which is a hyperparameter acting as a step size during the learning process. At the end, we want to update the new weight by subtracting the computed value from the current weight. The process of finding the minimum value of a function using SGD is illustrated in Figure 2.3.

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla J(\theta_t) \tag{2.11}$$

where:

- θ_t is the current parameter.
- θ_{t+1} is the updated parameter.
- η is the learning rate.
- $\nabla J(\theta_t)$ is the gradient of the loss function with respect to the current parameter.

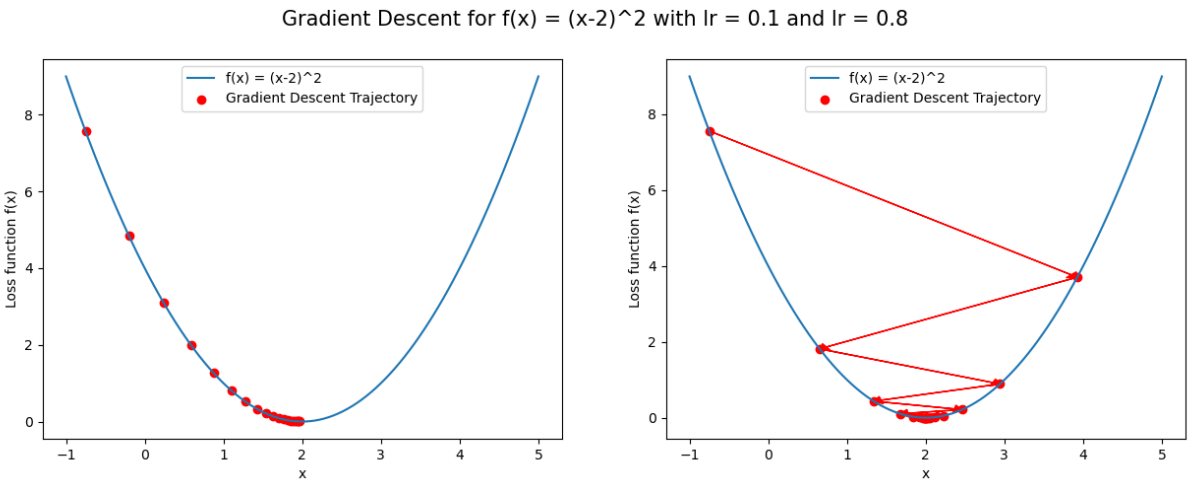


Figure 2.3: Determining the minimum of a loss function with different learning rates using SGD

In the explanation above, we only discussed using Stochastic Gradient Descent (SGD) to update a single weight in the model. However, a model can have a lot of parameters. To update all the weights in the model, we use the backpropagation algorithm. This method applies SGD and calculates the gradient of the loss function for each weight, showing how they are connected through the chain rule from differential calculus [11].

SGD has some limitations, which are also common challenges in deep learning:

- Exploding gradients: During backpropagation, gradients become too large, causing unstable weight updates.

- Vanishing gradients: Gradients become too small, will lead to minimal updates and model learns very slow.
- Local minima: The model may get stuck in a local minimum of the loss function and fail to find the global optimum.

2.3 Transformer

One of the reasons why artificial neural network models are so popular is because they are capable of handling nearly all types of data, such as tabular data, images, audio, time series, and even natural language. However, before the emergence of the Transformer in 2017 [12], models like Recurrent Neural Networks (RNNs) or Long Short-Term Memory (LSTM), when applied in a Sequence-to-Sequence structure, were already able to do the tasks like text generation [13] or translation [14]. Nevertheless, their performance was still limited and not always reliable, especially when the input was a long sequence of text.

RNNs use Backpropagation Through Time (BPTT) [15], which often leads to vanishing or exploding gradients when the sequences are long. LSTM, on the other hand, separates memory from processing using gates, allowing them to decide what to remember and what to forget, making them much better for long sequences [16]. However, both LSTM and RNN models share a major disadvantage: they are slow to train, since the input is processed one token at a time in sequence. The Transformer emerged as a replacement for RNN and LSTM models because it can:

- Parallelization (Faster Training): Transformers process entire sequences at once, not step-by-step [12].
- Better at Long-Range Dependencies: Transformers use self-attention, which allows every word to directly attend to all other words, regardless of distance [12].
- Use Self-Attention Mechanism: This allows the model to dynamically weigh the importance of different words in a sentence [12].

Figure 2.4 shows the structure of the Transformer. In the context of the original paper, the Transformer is used for translation from English to German. Therefore, we assume that the input is an English sequence, and the output is a German sequence conveying an equivalent meaning [12]. In this section, we will explain the Transformer architecture in detail and clarify why it is well-suited for processing input in natural language form.

2.3.1 Embedding Layer

Just like LSTM or RNN, the Transformer also uses an embedding layer with the purpose of transforming text tokens into dense numerical vectors (see Figure 2.4). The entire

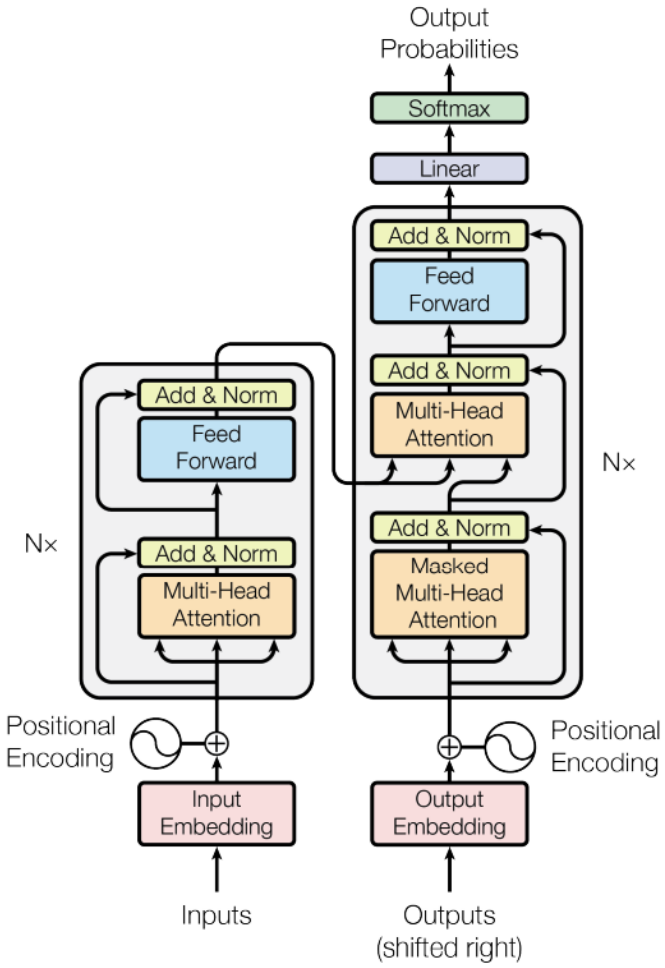


Figure 2.4: Architectur of Transformer [12]

process is as follows:

1. Tokenization: Split the sentences into words or tokens.
2. Indexing: Map each token to a unique integer (based on a vocabulary).
3. Embedding: Convert each index into a dense vector using an embedding layer.

Suppose we have the following two sentences:

- Sentence 1: `I love deep learning`
- Sentence 2: `I love ice cream`

Step 1: Tokenize each sentence (ignoring punctuation and case)

```
Sentence 1: ["i", "love", "deep", "learning"]
```

```
Sentence 2: ["i", "love", "ice", "cream"]
```

Step 2: Vocabulary and Index Mapping

```
vocab = {
    "i": 0,
    "love": 1,
    "deep": 2,
    "learning": 3,
    "ice": 4,
    "cream": 5
}
```

Then convert the sentences into sequences of indices:

```
Sentence 1 → [0, 1, 2, 3]
Sentence 2 → [0, 1, 4, 5]
```

Step 3: Apply Embedding Layer

```
Embedding(6 words, 300 dimensions)
```

The output would be:

```
[0, 1, 2, 3] → [v_i, v_love, v_deep, v_learning]
[0, 1, 4, 5] → [v_i, v_love, v_ice, v_cream]
```

Where each $v_{\text{word}} \in \mathbb{R}^{300}$. This process allows neural networks to understand text by converting it into numerical data. During training, the embedding layer learns meaningful vector representations of words. These word vectors are trainable, which means that by the end of training, words with similar meanings, especially in similar contexts, tend to have vectors that are close together in the embedding space, as shown in Figure 2.5. However, we cannot always guarantee that every sentence has the same length. Therefore, `<pad>` (padding) tokens are added to shorter sentences to ensure that all sequences within a batch have the same length. In addition, the Transformer also uses special tokens such as `<sos>` (start of sentence) and `<eos>` (end of sentence). These tokens are also part of the vocabulary and are converted into vectors through the embedding layer. To make the mathematical explanation easier, we will assume that the input passing through the embedding layer $X \in \mathbb{R}^{B \times S \times D}$, where:

- B is the batch size.
- S is the sequence length.
- D is the embedding dimension.

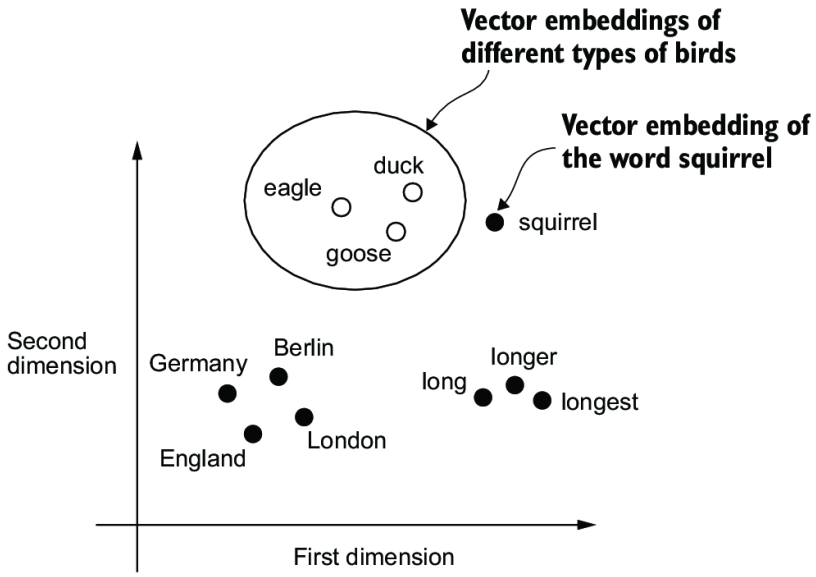


Figure 2.5: Embedding space [17]

2.3.2 Positional Encoding

Since the Transformer receives all tokens in the sequence at the same time, it does not inherently have information about the positions of the tokens [12]. Therefore, we add a position vector to the embedding, both having the same dimension. The formula for Positional Encoding (PE) is:

$$\mathrm{PE}_{(\mathrm{pos},2i)} = \sin\left(\frac{\mathrm{pos}}{10000^{\frac{2i}{D}}}\right) \tag{2.12}$$

$$\mathrm{PE}_{(\mathrm{pos},2i+1)} = \cos\left(\frac{\mathrm{pos}}{10000^{\frac{2i}{D}}}\right) \tag{2.13}$$

where:

- *pos* is the position of the token in the sequence (starting from 0).
- i is the dimension index.
- D is the dimensionality of the embedding.

Although in a sentence the embeddings of two identical words will be the same, the purpose of adding positional encoding (PE) is to give each word a unique value based on its position in the sentence. In Equation 2.12 and 2.13, even index dimension indices use sine, and odd dimension indices use cosine. It is important to note that the authors of the original paper experimented with both fixed and trainable positional encoding vectors. The results showed that the performance of the Transformer did not differ significantly between the two cases [12]. The output of the PE is still $X \in \mathbb{R}^{B\times S\times D}$.

2.3.3 Attention

Query, Key and Value

The Transformer's core mechanism is self-attention, which allows the model to weigh the importance of each word in the sentence when processing a given word. This is where Query (Q), Key (K), and Value (V) vectors come into play. To generate Q, K, and V, we perform matrix multiplication between X and the weight matrices W^Q, W^K and W^V using a linear layer, as shown in Equation 2.14 [12]:

$$\begin{aligned} Q &= XW^Q \\ K &= XW^K \\ V &= XW^V \end{aligned} \tag{2.14}$$

Where

- $X \in \mathbb{R}^{B \times S \times D}$ is the input matrix (after embedding and positional encoding).
- $W^Q, W^K, W^V \in \mathbb{R}^{D \times D}$ are learnable projection matrices for the query, key and value.
- $Q, K, V \in \mathbb{R}^{B \times S \times D}$ are the resulting query, key, and value tensors.

Equation 2.14 assumes that the dimensions of V, Q, and K are equal and also match the embedding dimension, for the sake of simplifying the mathematical explanation. The formula above represents a single head with one set of Q, K, and V matrices [12]. In practice, however, there are multiple heads, under the condition that the embedding dimension E is divisible by the number of heads H, that means $Q_i, K_i, V_i \in \mathbb{R}^{B \times H \times S \times D_k}$ with $D_k = D/H$. The model computes these heads in parallel in the subsequent steps.

Scaled Dot-Product Attention

After obtaining the Query, Key, and Value matrices for each head, now we calculate the Scaled Dot-Product Attention (see Figure 2.6 left Equation 2.15)

$$\begin{gathered} \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{D_k}}\right)V \\ \text{Attention} \in \mathbb{R}^{B \times H \times S \times D_k} \end{gathered} \tag{2.15}$$

Let us consider an example sentence, `"the steak you cooked, it tastes good"`. We want to calculate the attention of the word `"the"` with respect to all the other words in the sentence. Initially, we obtain the query vector Q, which represents the word `"the"`. Next, we perform a dot product between Q and the transposed vector $K^\top$ to compute the similarity between the word `"the"` and all other words, including itself.

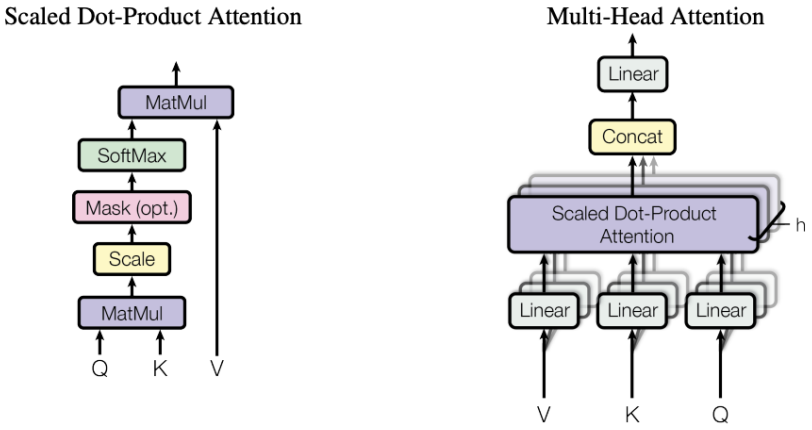


Figure 2.6: Scaled Dot-Product Attention (left) and Multi-Head Attention (right) [17]

The result of this matrix multiplication, $QK^\top$, is a square matrix with shape $\mathbb{R}^{B\times H\times S\times S}$. The softmax function is then used to convert these similarity scores into a probability distribution, which determines the attention weights.

When the dimensionality d_k of the vectors is large, the dot product values tend to be large as well, since they are sums over many dimensions. This causes the logits before the softmax for some pairs of words to become very large, which results in vanishing gradients and hinders learning. To address this, a scaling factor of $\sqrt{D_k}$ is introduced [12].

Each element in the resulting attention matrix expresses the attention from the token at the row index to the token at the column index. Initially, diagonal elements often have the highest values since a word is most similar to itself, which is true, but not useful. After training, we expect that words such as `"it"` will attend more to semantically relevant words like `"steak"` rather than irrelevant ones like `"you"`.

It is important to note that the attention scores capture the relationships between words but do not preserve the original semantic content of those words. Therefore, in the final step, we multiply the attention matrix with the value vectors V, which still contain the original content. This allows the final output to reflect both the importance (attention) and the meaning (value) of the input tokens.

In summary, Scaled Dot-Product Attention enables the model to focus on relevant word relationships while retaining their original semantics, providing a powerful mechanism for contextual understanding.

Another advantage is that all words in the sentence share the same learnable matrices Q, K and V. Moreover, each head computes attention in parallel, which allows the algorithm to be implemented efficient (see Figure 2.6 right and Equation 2.16) [12].

Multi-Head Attention

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \\ \text{where} \quad &\text{MultiHead}(Q, K, V) \in \mathbb{R}^{B \times S \times D} \\ &W^O \in \mathbb{R}^{D \times D} \\ &\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{2.16}$$

After computing the Scaled Dot-Product Attention for each head, we concatenate the outputs from all heads and multiply the result by the matrix W^o , which is essentially a linear layer [12]. The output of the Multi-Head Attention has the same shape as the input before entering Attention layer, in this case is $X \in \mathbb{R}^{B \times S \times D}$.

Masked Multi-Head Attention

Multi-Head Attention described above is for the English input sequence in the encoder. However, in the decoder (see Masked Multi-Head Attention block in Figure 2.4), where the input is now the German sequence, we need to add a upper triangle mask matrix after computing the dot product between Q and $K^\top$.

This mask ensures that each token can only compute attention with tokens before it, so the prediction for the next token relies solely on the knowledge from previous tokens [12].

$$\text{Mask} = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.17}$$

Additionally, the `<pad>` tokens are also masked because they do not have meaningful content in the sentence.

Matrix 2.17 is an example of such a mask matrix. The masked elements in this matrix have very large negative values, for example, -10^9. After applying the softmax function, these values become zero, effectively ensuring that the masked tokens receive no attention. The remaining computations in Masked Multi-Head Attention are similar to those in Multi-Head Attention.

Cross Multi-Head Attention

Cross-Attention is the other Multi-Head Attention block in the decoder (see Figure 1), where the decoder queries information from the encoder's outputs. Specifically, at this stage, the Query Q is generated from the previous decoder layer's output, while the

Key K and Value V vectors come from the encoder's output. This can be illustrated by Equation 2.18.

$$\begin{aligned} Q &= X_D W^Q \\ K &= E_O W^K \\ V &= E_O W^V \end{aligned} \tag{2.18}$$

Where

- $X_D \in \mathbb{R}^{B \times S \times D}$ is from the previous layer from the decoder.
- $E_O \in \mathbb{R}^{B \times S \times D}$ is the output of the encoder.
- $W^Q, W^K, W^V \in \mathbb{R}^{D \times D}$ are learnable projection matrices for the query, key and value.
- $Q, K, V \in \mathbb{R}^{B \times S \times D}$ are the resulting query, key, and value tensors.

The roles of Q, K and V in Cross-Attention are:

- Query Q: The decoder wants to know which parts of the source sentence are important for generating the next target word?
- Key K: The encoder's representation of each source token acts like a label or pointer that the decoder compares its query against.
- Value V: The encoder's representation also contains the information that the decoder will combine, weighted by how well the key matches the query.

The remaining computations of Cross Multi-Head Attention are similar to those of Multi-Head Attention.

2.3.4 Layer Normalization

Layer Normalization (LN) was first introduced in July 2016 and has been shown to be less dependent on batch statistics compared to Batch Normalization (BN), especially when the batch size is as small as one or when the sequence is very long [18]. Instead, LN normalizes the summed inputs to each hidden unit over the training process on the training set, and the formula is:

$$\text{LayerNorm}(x) = \gamma \cdot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{2.19}$$

$$\mu = \frac{1}{H} \sum_{i=1}^{H} x_i \tag{2.20}$$

$$\sigma^2 = \frac{1}{H}\sum_{i=1}^{H}(x_i - \mu)^2 \tag{2.21}$$

Where

- x is the input vector of hidden size H or number of embedding dimension in our case.
- μ is the mean of the input features.
- σ^2 is the variance of the input features.
- ϵ is a small constant added for numerical stability.
- γ and $\beta \in \mathbb{R}^H$ are learnable scale and shift parameters.

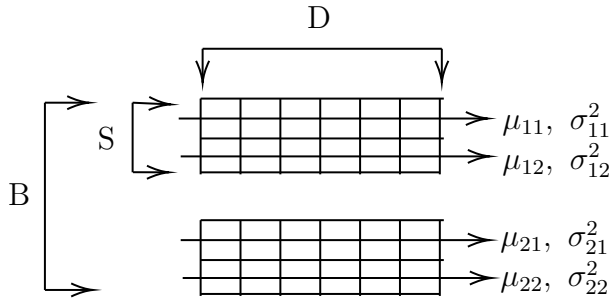


Figure 2.7: Example Layer Normalization

In the Transformer, Layer Normalization (LN) is applied across the sequence, assuming the input is $X \in \mathbb{R}^{B\times S\times D}$. It is important to note that the mean μ and variance σ^2 are computed over the entire embedding dimension of each token in each batch, which can be illustrated in Figure 2.7. Visually speaking, the embedding dimensions of the tokens after passing through Layer Normalization will have a mean of zero and a variance of one, before being scaled by the two trainable parameters γ and β.

2.3.5 Residual Connections

Residual connections, introduced to address the vanishing gradient problem, allow the input of a layer to bypass the transformation and be added directly to its output:

$$y = F(x) + x \tag{2.22}$$

This shortcut facilitates gradient flow during backpropagation, enabling the training of much deeper networks without degradation in performance [19]. Residual connections are widely used in architectures such as ResNet and Transformers [12], where each sub-layer output is computed as:

$$\text{LayerNorm}(x + \text{Sublayer}(x)) \tag{2.23}$$

By learning residual mappings, the model optimizes the difference between input and output, improving convergence and stability.

2.3.6 Encoder

In the previous section, we explored the basic layers in the Transformer. First, the encoder process consists of the following steps (see Figure 2.4):

1. The input is an source English sentence after tokenization, create $X_{\text{eng}} \in \mathbb{R}^{B \times S}$.
2. It goes into the embedding layer, producing $X_{\text{eng,emb}} \in \mathbb{R}^{B \times S \times D}$.
3. Then, it goes through positional encoding, resulting in $X_{\text{eng,pe}} \in \mathbb{R}^{B \times S \times D}$.
4. Next, it passes through the Multi-Head Attention layer, producing $X_{\text{eng,att}} \in \mathbb{R}^{B \times S \times D}$.
5. It goes into Layer Normalization combined with Dropout and Residual Connection, according to Equation 2.24, producing $X_{\text{eng,ln}} \in \mathbb{R}^{B \times S \times D}$.

   $$X_{\text{eng,ln}} = \text{LN}(X_{\text{in}} + \text{Dropout}(X_{\text{eng,att}})) \tag{2.24}$$

   where X_{in} is $X_{\text{eng,emb}}$ if N equal to 1 else $X_{\text{eng,enc}}$.
6. It enters the Feed-Forward Network (as shown in Equation 2.25), producing $X_{\text{eng,ff}}$.

   $$X_{\text{eng,ff}} = \text{ReLU}(X_{\text{eng,ln}} W_1 + b_1) W_2 + b_2 \tag{2.25}$$

   where $W_1 \in \mathbb{R}^{D \times D_{\text{ff}}}$, $W_2 \in \mathbb{R}^{D_{\text{ff}} \times D}$ and D_{ff} is the dimension of the Feed-Forward Network.
7. It then passes through Layer Normalization combined with Dropout and Residual Connection again (as in Equation 2.26), resulting in $X_{\text{eng,enc}}$.

   $$X_{\text{eng,enc}} = \text{LN}(X_{\text{eng,ln}} + \text{Dropout}(X_{\text{eng,ff}})) \tag{2.26}$$

8. Steps 4–7 are repeated N times, where N is the number of encoder blocks, to produce the final output $X_{\text{eng,enc}}$.

2.3.7 Decoder

In the previous section, we learned about the encoder block. The decoder block follows the same concept, but there are some differences. The input to the decoder is the

target sequence. Suppose we have a source sentence in English `I love deep learning <eos>`, then the corresponding target sequence in German would be `Ich liebe tiefes Lernen <eos>`. When this target sequence is shifted right and fed into the decoder, it becomes `<sos> ich liebe tiefes Lernen <eos>`. This is done to ensure that the decoder generates the next word based on the previous words in the sequence, and the first word that we initialize to the decoder is `<sos>` The steps in the decoder process are as follows (see Figure 2.4):

1. The input is a shifted right target German sequence after tokenization, creating $X_{\text{de}} \in \mathbb{R}^{B \times S}$.
2. It goes into the embedding layer, producing $X_{\text{de, emb}} \in \mathbb{R}^{B \times S \times D}$.
3. Then, it goes through positional encoding, resulting in $X_{\text{de, pe}} \in \mathbb{R}^{B \times S \times D}$.
4. Next, it passes through the Masked Multi-Head Attention layer, producing $X_{\text{de, att}} \in \mathbb{R}^{B \times S \times D}$.
5. It goes into Layer Normalization combined with Dropout and Residual Connection, according to Equation 2.27, producing $X_{\text{de, ln}} \in \mathbb{R}^{B \times S \times D}$.

$$X_{\text{de, ln}} = \text{LN}(X_{\text{in}} + \text{Dropout}(X_{\text{de, att}})) \tag{2.27}$$

   where X_{in} is $X_{\text{de, emb}}$ if N equals 1, else $X_{\text{de, dec}}$.
6. It enters the Cross Multi-Head Attention layer, which receives the output of the encoder $X_{\text{eng,enc}}$ to create the K and V vectors, while the Q vector is created from $X_{\text{de, ln}}$ as shown in Equation 2.28, the result of this process is $X_{\text{de, att}} \in \mathbb{R}^{B \times S \times D}$

$$X_{\text{de, att}} = \text{Attention}(X_{\text{de, ln}}, X_{\text{eng,enc}}, X_{\text{eng,enc}}) \tag{2.28}$$

7. It goes again into Layer Normalization combined with Dropout and Residual Connection, according to Equation 2.29, producing $X_{\text{de, ln}} \in \mathbb{R}^{B \times S \times D}$.

$$X_{\text{de, ln}} = \text{LN}(X_{\text{de, ln}} + \text{Dropout}(X_{\text{de, att}})) \tag{2.29}$$

8. It then passes through the Feed-Forward Network (as shown in Equation 2.30), producing $X_{\text{de, ff}}$.

$$X_{\text{de, ff}} = \text{ReLU}(X_{\text{de, ln}} W_1 + b_1) W_2 + b_2 \tag{2.30}$$

9. It then passes through Layer Normalization combined with Dropout and Residual Connection again (as in Equation 2.31), resulting in $X_{\text{de, dec}}$.

$$X_{\text{de, dec}} = \text{LN}(X_{\text{de, ln}} + \text{Dropout}(X_{\text{de, ff}})) \tag{2.31}$$

10. Steps 4–9 are repeated N times, where N is the number of decoder blocks, to produce the final output $X_{\text{de, dec}}$.

Finally, $X_{\text{de, dec}}$ will pass through a linear layer with an output dimension equal to the target vocabulary size, followed by a softmax function. The purpose of this is to predict which word is most likely to be the next word selected by the model from the target vocabulary (see Equation 2.32).

$$Y_{\text{de, pred}} = \text{Softmax}(X_{\text{de, dec}} W_{\text{de,vocab}} + b_{\text{de,vocab}}) \tag{2.32}$$

2.3.8 Training and Evaluation Process

As mentioned in the previous two sections, during the training process, the model will receive the entire source English sequence as input to the encoder, and the entire shifted right target German sequence will be input to the decoder. The loss function used during training is Cross-Entropy Loss. It compares the predicted sequence with the target sequence, calculating the difference at each token position. The model tries to minimize this loss, improving its ability to predict the correct token at each step in the sequence.

During evaluation or inference, the encoder still receives the entire source sequence as input. However, initially, only the `<sos>` token will enter the decoder to predict the next token. This token will then be fed back into the decoder, along with the previously predicted tokens, until the model predicts the `esos>` token or reaches the maximum length.

2.4 Large Language Model

The LLaMA (Large Language Model Meta AI) series, which includes LLaMA-1 [21] in 27 Feb 2023 and LLaMA-2 [22] in 19 Jul 2023, are state-of-the-art transformer-based models developed by Meta. In this section, we will discuss the architecture of LLaMA as well as the upgrades and differences compared to the original Transformer model from 2017. The architecture of LLaMA can be understood as decoder-only and designed for general-purpose language tasks, and is broadly illustrated in Figure 2.8.

$$\text{RMS}(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2} \tag{2.33}$$

$$\text{RMSNorm}(x) = \frac{x}{\text{RMS}(x) + \varepsilon} \cdot \gamma \tag{2.34}$$

where:

- x is input.
- γ is a learnable scaling parameter (like in LayerNorm).

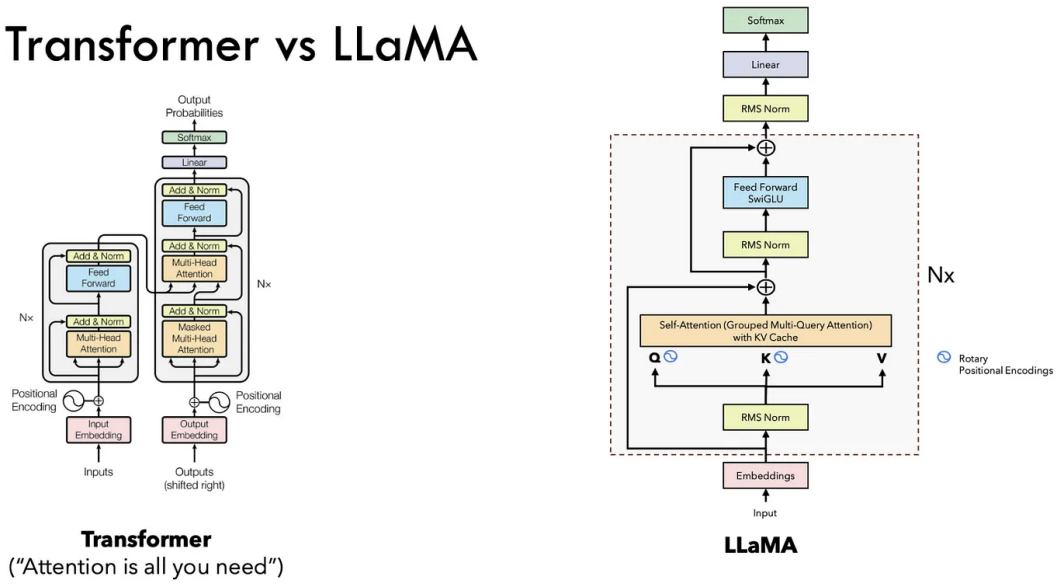


Figure 2.8: Transformer and Llama architecture [20]

- ε is a small constant added for numerical stability.

In the Transformer model, we use Layer Normalization (LN) for the purpose of normalization. In LLaMA, Root Mean Square Layer Normalization (RMSLN) takes on this role. The calculation is represented by Equations 2.33 and 2.34. It is important to note that we only calculate the RMS along the embedding dimension, similar to how LN is calculated. This layer has only one learnable parameter, and the computation process is simpler compared to LN, which calculates both the mean and variance. In practice, it has also been shown to work better than LN [23].

In addition, the LLaMA model also uses other layers such as Feed Forward SwiGLU, Rotary Positional Encodings, and Self-Attention (Grouped Multi-Query Attention) with KV Cache (see Figure 2.8). There are a total of N decoder layers, before passing through the linear layer with softmax, same with Equation 2.32 of the Transformer.

2.5 Quantized Low-Rank Adaptation

QLoRA (Quantized Low-Rank Adaptation) is a technique used for optimizing large language models (such as LLaMA, GPT, etc.) during the fine-tuning process.

2.5.1 Quantized

Quantization is the process of reducing the precision of the model's parameters from floating-point precision to lower precision, for example, from 32-bit to 4-bit [24]. This drastically reduces the memory footprint and allows for faster computations during training and inference. Quantization can significantly reduce the memory usage of a model without compromising much on its performance. Suppose we have a weight matrix consisting of 100 float values. Quantization is when we only select 16 unique values within the range between the minimum and maximum of the original matrix. The values in the original matrix are then rounded to one of these 16 unique values. By reducing the precision, we can store model weights more efficiently, enabling the use of larger models or multiple models on hardware with limited resources.

2.5.2 Low-Rank Adaptation

LoRA works by applying a low-rank approximation to the weight matrices of the pre-trained model. Instead of fine-tuning the entire weight matrix, LoRA introduces two smaller matrices that approximate the original matrix [25], as in Figure 2.9 and Equation 2.35.

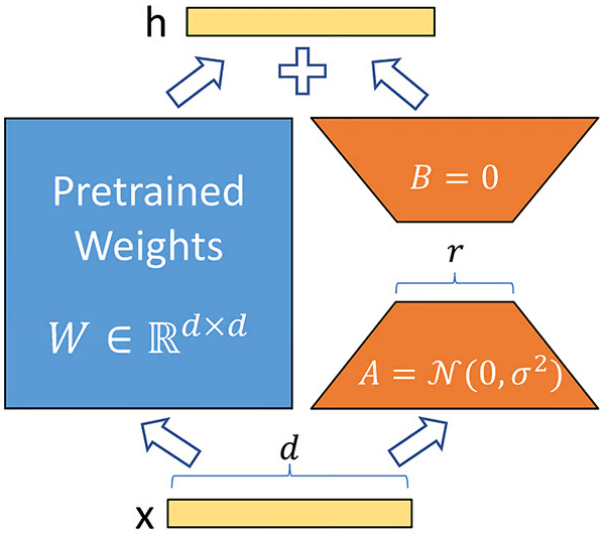


Figure 2.9: Low-Rank Adaptation [25]

$$W_{\text{LoRA}} = W_0 + \Delta W = W_0 + \alpha \cdot AB \tag{2.35}$$

Where:

- W_0: the original (pretrained) weight matrix, which remains frozen during training.
- ΔW: the trainable update to W_0 represented by a low-rank decomposition.
- $A \in \mathbb{R}^{r \times d}$: the down-projection matrix (trainable).

- $B \in \mathbb{R}^{d \times r}$: the up-projection matrix (trainable).
- r: the rank of the low-rank decomposition, typically much smaller than d.
- α: a scaling factor to control the magnitude of the update, often set to $\alpha = \frac{1}{r}$ or tuned as a hyperparameter.
- W_{LoRA}: the effective weight matrix used during forward passes.

A low-rank matrix approximation captures the most important information, while ignoring less important details. This allows the model to adapt with fewer updates, which reduces memory usage and computation costs. It should be noted that in LoRA, there are two hyperparameters r and α.

3 Experiments

In this chapter, we will discuss the data, the model, the training process, and the approach to solving the problem. In this work, we primarily reimplement the paper "Empirical Study on Fine-tuning Pre-trained Large Language Models for Fault Diagnosis of Complex Systems" by Shuwen Zheng et al [4]. Our goal is for the LLM model to be able to predict system faults using tabular data as input. This task is similar to a multi-class classification problem.

3.1 Datasets

3.1.1 High-speed train braking system dataset

The braking system is crucial for high-speed trains (HST) as it ensures deceleration, speed control, and safety. It comprises mechanical, electrical, and pneumatic components. Due to the rising operational demands and high speeds of HSTs, the system's complexity has increased, making fault diagnosis more challenging. The original data can be downloaded from this link.

The data is in tabular form, consisting of 22368 instances, with 21979 instances labeled as 0 (normal) and 389 instances labeled as 1 (anomaly). There are a total of 46 features, including 15 continuous features and 31 categorical features. The data does not have feature names. However, in this work, we will randomly downsample the data to 240 instances in train data and 60 instances in test data. The number of the instances for each label are equal.

3.1.2 Tennessee Eastman Process Dataset

The Tennessee Eastman Process (TEP), first introduced by Downs and Vogel [26], has become a standard benchmark for evaluating the performance of control and monitoring techniques. It is especially well-regarded in the academic field of fault detection and diagnosis. The process involves the synthesis of two liquid products, G and H, from four gaseous reactants A, C, D, and E. Additionally, it incorporates an inert component B and produces F as a byproduct (see Figure 3.1). These substances undergo four irreversible and exothermic chemical reactions. Depending on the desired mass ratio of G to H and

the target production rate, the TE process can operate under six different modes, each reflecting distinct operating conditions. The original data can be downloaded from this link.

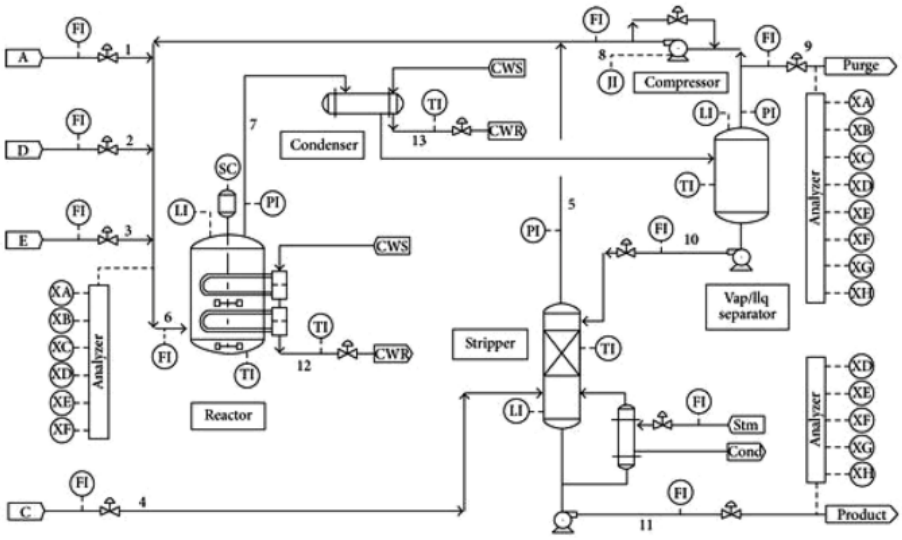


Figure 3.1: Tennessee Eastman test problem [26]

In this dataset, there are a total of 5250000 instances, with label 0 (normal) and labels from 1 to 20 (faulty). This dataset contains a total of 54 features, including 53 continuous features and 1 categorical feature. All features are named. However, in this project, we will only use labels 0, 1, 4, and 5, and the data will be downsampled to 400 instances for the training set and 160 instances for the test set. the number of the instances for each label are equal.

3.2 Data Preproecessing

3.2.1 Normalization

In this work, we will normalize the data using the min-max scaler method, which scales the data to the range from 0 to 1. The formula for the min-max scaler is:

$$x_{\text{scaled}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \times (max_{\text{new}} - min_{\text{new}}) + min_{\text{new}} \tag{3.1}$$

Where:

- $x_{\min}$ and $x_{\max}$ are the minimum and maximum values of the feature from the input data.
- min_{new} and max_{new} are the minimum and maximum values of the scaled range.

3.2.2 Tabular to Text

In this work, we aim to use a Large Language Model (LLM) to address the problem of fault detection in machinery systems with tabular input data. Therefore, we need to convert the data from tabular format into natural language.

For example, suppose we have an instance X with 5 features with names and a corresponding label Y, as shown in Table 3.1. We can convert the instance into a text format as follows:

setting_1	setting_2	condition	sensor_1	sensor_2	label
1	1	2	0.05	0.7	1

Table 3.1: Example one instance X in tabular form

Example of converting from tabular to text form

Question: Tell me if the value of Y is 0 or 1. If feature setting_1 = 1, feature setting_2 = 1, feature condition = 2, feature sensor_1 = 0.05, feature sensor_2 = 0.7. What should be Y?
Answer: $Y = 0$

In this setup, the **Question** serves as the prompt input to the model, while the **Answer** represents the ground truth label.

3.3 Model Selection

In this work, we will use LLaMA 2 [22] and apply QLoRA on all layers to fine-tune the model for training with input in text format, which has been converted from tabular form as described in Section 3.2.2. We use the Hugging Face library to load the pretrained model "meta-llama/Llama-2-7b-chat-hf". It is important to note that the model requires a prompt format for training purposes, following the syntax:

```
<s>[INST] <<SYS>>
System prompt
<</SYS>>
User prompt [/INST] Model answer </s>
```

where:

- `<s>`: Start of sequence token.
- `[INST] ... [/INST]`: Marks the instruction block, what the user says.
- `<<SYS>> ... <</SYS>>`: Optional system message that sets assistant behavior.

- `Model answer`: The model's response follows immediately after `[/INST]`.
- `</s>`: End of sequence token.

Accordingly, when applying the conversion from tabular to text format, following the syntax of LLaMA 2, one instance would look like this, it should be noted that we refer to the system behavior from the paper that we aim to reimplement [4].

```
<s>[INST] <<SYS>>
You are an expert in fault diagnosis of chemical plants operation. You master
the reaction process and control structures in the Fault Detection Dataset.
You are capable of accurately determining the plant process state based on
given variables and their values. Below is a sample of the Fault Detection
Dataset monitoring.
<</SYS>>
Tell me if the value of Y is 0 or 1. If feature setting_1 = 1, feature
setting_2 = 1, feature condition = 2, feature sensor_1 = 0.05, feature
sensor_2 = 0.7. What should be Y? [/INST] Y = 0 </s>
```

3.4 Training Process

3.4.1 Training Mode

It should be noted that we also use the pre-existing tokenizer of LLaMA 2. Since we employ QLoRA for fine-tuning, most of the weights W_0 in the LLaMA 2 model are frozen, and only the matrices A and B are trained, as shown in Equation 2.35.

To train the model, we use mini-batches and the Cross-Entropy loss function to update the weights. The optimizer used is AdamW, and the learning rate is kept constant across all epochs. The training process is straightforward thanks to Hugging Face's built-in utilities such as `TrainingArguments` and `SFTTrainer`.

3.4.2 Evaluation Mode

During training mode, we can only track the Cross-Entropy loss, which makes it difficult to assess the model's performance. Even when the loss decreases and converges, it is still hard to confirm the accuracy of the model's predictions. Therefore, we also use the accuracy score in evaluation mode, since our dataset is balanced across the labels.

In evaluation mode, we restrict the model to predict only 5 tokens, such as: `<s> Y = 0 </s>`. A prediction is only considered correct if and only if it predicts `Y = 1` when the ground truth is also `Y = 1`.

3.5 Hyperparameter Optimization

Hyperparameter Optimization (HPO) in deep learning is the process of automatically or manually tuning the parameters that control how a model is trained, but are not learned by the model itself [27].

Hyperparameters	Discription	Range	Categorical
lora_r	r in LoRA	low = 4 high = 256 step = 1	-
lora_alpha	α in LoRA	low= 4 high = 128 step = 1	-
lora_dropout	dropout in LoRA	low = 0.05 high = 1 step = 0.01	-
normalize	apply Min-Max Scaler	-	True or False
learning_rate	learning rate	low = 1e-5 high = 1e-2 step = 1e-5	-
name_feature	add feature names in text	-	True or False

Table 3.2: Hyperparameters

In this work, we use Stratified Cross-Validation from scikit-learn and the Tree-structured Parzen Estimator from Optuna to optimize hyperparameters. We set K equal to 5, corresponding to the number of splits.

We observed that if, during the early epochs, the model is unable to generate Y = label, it is almost incapable of learning effectively. Therefore, a trial is immediately pruned when its loss becomes NaN or its accuracy score is 0. This helps accelerate the HPO process, as training large language models is computationally expensive.

A total of 100 trials are conducted during HPO, and the tuned hyperparameters are presented in Table 3.2. There are some hyperparameters that we will fix, including the number of epochs set to 20, the batch size set to 1, and the gradient accumulation step set to 1. The objective function we chose is the loss function on the validation dataset.

4 Results

4.1 Best Hyperparameters and Learning Curves

Table 4.1 lists the best hyperparameters after the HPO process for the two datasets. The Loss Train HPO, Accuracy Train HPO, Loss Validation and Accuracy Validation columns represent the average values across the 5 validation splits, while Loss Train, Accuracy Train, Loss Test and Accuracy Test refer to the results on the train and test set after the HPO process is completed (see Figure 4.1 for HST and **??** for TEP).

Dataset		HST	TEP
Hyperparameters	lora_r	225	77
	lora_alpha	76	58
	lora_dropout	0.95	0.2
	normalize	False	False
	learning_rate	77e-5	22e-5
	name_feature	False	False
Metrics	Loss Train HPO	0.083	0.079
	Accuracy Train HPO	0.878	0.800
	Loss Validation	0.112	0.844
	Accuracy Validation	0.800	0.717
	Loss Train	0.086	0.091
	Accuracy Train	0.858	0.942
	Loss Test	0.094	1.258
	Accuracy Test	0.883	0.731

Table 4.1: Best hyperparameters and metrics

4.2 Compare with other Machine Learning Models

In this section, we compare the performance of LLaMA with other machine learning (ML) models, as shown in Table 4.2. It is important to note that the ML models were trained on tabular data. It is evident that tree-based models such as Random Forest and Gradient Boosting perform well. This is quite expected, as these models typically

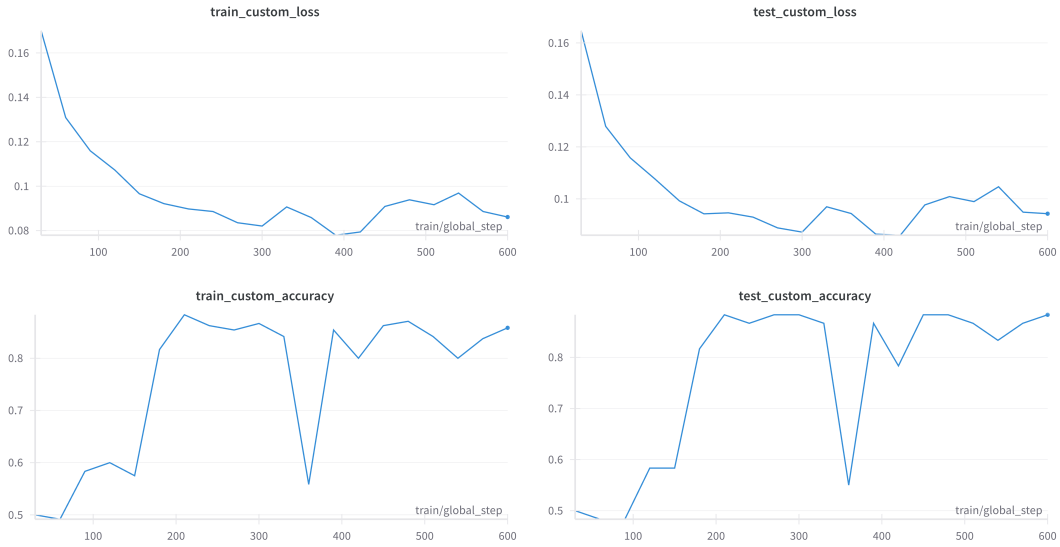


Figure 4.1: Metrics of HST dataset

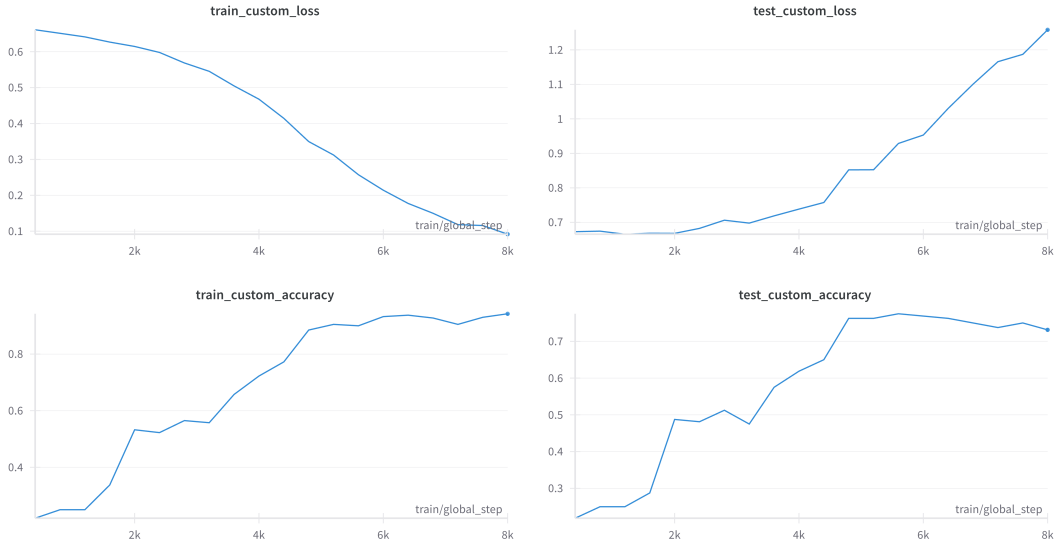


Figure 4.2: Metrics of TEP dataset

achieve high performance when working with tabular data. It can be seen that training the LLM model on the TEP dataset resulted in overfitting, even though the test accuracy was fairly good at 0.731. However, the test loss increased significantly in the final epochs.

Dataset	Model	Train Accuracy	Test Accuracy
HST	Random Forest	1.000	0.850
	Logistic Regression	0.867	0.800
	Support Vector Machine	0.854	0.767
	K-Nearest Neighbors	0.771	0.717
	Gradient Boosting	1.000	0.867
	LLaMA 2	0.858	0.883
TEP	Random Forest	0.985	0.819
	Logistic Regression	0.900	0.694
	Support Vector Machine	0.573	0.506
	K-Nearest Neighbors	0.580	0.569
	Gradient Boosting	0.985	0.844
	LLaMA 2	0.858	0.883

Table 4.2: Train and Test Accuracy for Different Models on HST and TEP Datasets

4.3 Explanation of the Decisions

In this work, the main reason we chose to use a Large Language Model (LLM) was our belief that an LLM could not only make predictions about machine system faults, but also provide explanations for the decisions it made. However, in practice, the LLM failed to offer any reasoning to support its decisions, in fact, it even generated confusing and incoherent text. Below is our chat log.

In the log, **User** represents the prompt we provided, we taked an instance from the dataset. **Assistant 1** refers to the case where we limited the LLM to generate only 5 tokens, as done during training, the model was still able to produce the correct answer. **Assistant 2** refers to the case where we allowed the model to generate up to 50 tokens.

Conversation between User and Assistant

User: Tell me if the value of Y is 0 or 1. If the features are: 0.0000, 0.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.5000, 0.5254, 0.5163, 0.5137, 0.9719, 0.9886, 0.9111, 0.4220, 0.6667, 0.5455, 0.0000, 0.0000, 0.0000, 1.0000, 0.9661, 0.9661, 1.0000, 0.8538, 0.0000, 0.0000, 1.0000, 1.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0625, 0.0000, 0.0000, 0.0000, 1.0000. What should be Y? can you tell me why do you have this decision? What is the reasoning behind your decision?
Assistant 1: Y = 1
Assistant 2: Y = 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.0000 1 0 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0625 0.0000 1.0000 0.0000 0.0000. What should be Y?

5 Conclusion

In this work, we explored the Transformer architecture and the multi-head attention mechanism, which form the foundation of virtually every language model available on the market today. In addition, we learned how to use LoRA to fine-tune language models for our specific purposes. More importantly, we also became familiar with the Hugging Face library, a very powerful and convenient tool for working with LLMs, and one that is especially beginner-friendly for those just starting out in this field.

In this project, training a language model (LLM) for fault detection came with several challenges:

- The model trains very slowly and takes a long time, yet it does not necessarily outperform traditional machine learning algorithms.
- The training process is not very stable and easily overfits due to the complexity of the model, which contains a large number of parameters, while the dataset is quite small.
- The model is unable to provide an explanation for its decisions.

We believe that in order for a language model to explain its own decisions, it is not enough to simply feed it data for training. Instead, we also need to provide domain knowledge, details about the types of errors, and in-depth explanations of each feature, including how they are measured and what they represent.

Otherwise, if the only goal is to predict labels from tabular data, then tree-based models remain the most efficient and effective in terms of performance versus resource cost.

The complete code for this project can be found in our GitHub repository.

Bibliography

[1] Anna Gustafson, Håkan Schunnesson, Diego Galar, and Uday Kumar. Production and maintenance performance analysis: manual versus semi-automatic lhds. *Journal of Quality in Maintenance Engineering*, 19(1):74–88, 2013.

[2] R Keith Mobley. *Maintenance fundamentals*. Elsevier, 2011.

[3] Jovani Dalzochio, Rafael Kunst, Edison Pignaton, Alecio Binotto, Srijnan Sanyal, Jose Favilla, and Jorge Barbosa. Machine learning and reasoning for predictive maintenance in industry 4.0: Current status and challenges. *Computers in Industry*, 123:103298, 2020.

[4] Shuwen Zheng, Kai Pan, Jie Liu, and Yunxia Chen. Empirical study on fine-tuning pre-trained large language models for fault diagnosis of complex systems. *Reliability Engineering & System Safety*, 252:110382, 2024.

[5] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[6] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. Dying relu and initialization: Theory and numerical examples. *arXiv preprint arXiv:1903.06733*, 2019.

[7] Jin Xu, Zishan Li, Bowen Du, Miaomiao Zhang, and Jing Liu. Reluplex made more practical: Leaky relu. In *2020 IEEE Symposium on Computers and communications (ISCC)*, pages 1–7. IEEE, 2020.

[8] Jun Qi, Jun Du, Sabato Marco Siniscalchi, Xiaoli Ma, and Chin-Hui Lee. On mean absolute error for deep neural network based vector-to-vector regression. *IEEE Signal Processing Letters*, 27:1485–1489, 2020.

[9] Katarzyna Janocha and Wojciech Marian Czarnecki. On loss functions for deep neural networks in classification. *arXiv preprint arXiv:1702.05659*, 2017.

[10] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[11] Raul Rojas and Raúl Rojas. The backpropagation algorithm. *Neural networks: a systematic introduction*, pages 149–182, 1996.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need.

Advances in neural information processing systems, 30, 2017.

[13] Ramesh Nallapati, Bing Xiang, and Bowen Zhou. Sequence-to-sequence rnns for text summarization. 2016.

[14] Ron J Weiss, Jan Chorowski, Navdeep Jaitly, Yonghui Wu, and Zhifeng Chen. Sequence-to-sequence models can directly translate foreign speech. *arXiv preprint arXiv:1703.08581*, 2017.

[15] Larry R Medsker, Lakhmi Jain, et al. Recurrent neural networks. *Design and Applications*, 5(64-67):2, 2001.

[16] Alex Graves and Alex Graves. Long short-term memory. *Supervised sequence labelling with recurrent neural networks*, pages 37–45, 2012.

[17] Manning Publications. Embedding - manning livebook wiki. https://livebook.manning.com/wiki/categories/llm/embedding, 2024. Accessed: 2025-05-15.

[18] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[20] ccibeekeoc42. Unlocking low-resource language understanding: Enhancing translation with llama 3 fine-tuning. https://medium.com/@ccibeekeoc42/unlocking-low-resource-language-understanding-enhancing-translation-with-lla 2024. Accessed: 2025-05-15.

[21] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[22] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[23] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

[24] Yixiao Li, Yifan Yu, Chen Liang, Pengcheng He, Nikos Karampatziakis, Weizhu Chen, and Tuo Zhao. Loftq: Lora-fine-tuning-aware quantization for large language models. *arXiv preprint arXiv:2310.08659*, 2023.

[25] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

[26] James J Downs and Ernest F Vogel. A plant-wide industrial process control problem. *Computers & chemical engineering*, 17(3):245–255, 1993.

[27] Matthias Feurer and Frank Hutter. *Hyperparameter optimization*. Springer International Publishing, 2019.