Lightweight Fine-Tuning of Large Language Models To Be Used In Cloud Applications in Document Manipulation.

Gabriele Monti September 2024

Important notes to the reader: While you will find all the code in the Github repository of the project, you should be aware that Colab notebooks are not rendered in Github, this means that you need to download them and then reupload them in your Colab.

Otherwise the best way is to access them via the links provided in the research below. Alternatively, you can open the GitHub URL in an incognito Chrome session. This method seems to render the Colab notebooks.

All the code present in this research is available at the following github repository https://github.com/Birkbeck/msc-projects-2023-4-Gabriele_Monti_PEFT

Chapter 1 Introduction.

This project focuses on techniques and a hands-on approach to **efficiently fine-tuning large language models (LLMs)**. This simply means that we can take a generic pretrained model, integrate additional knowledge and conditions its output for our specific downstream task. Firstly the inner workings of the **transformer architecture** (as explained in the paper **Attention Is All You Need**) will be discussed in detail. The project will cover **PEFT** (**Parameter-Efficient Fine-Tuning**) techniques. It will explore various efficient fine-tuning techniques that enable the transfer of knowledge from generic pre-trained models to task-specific models. In simple words these are techniques that aim to reduce the complexity of certain structures (attention

mechanism layers) in the models. By adopting these steps, the training process will be **significantly faster** without any noticeable loss in output quality
Finally a hand ons experiment will be conducted to create a model able to **extract structured details from resumes**. This experiment aims to demonstrate that fine-tuning LLMs can significantly reduce costs for Human Resources departments by automating the resume screening process.
The obtained model will be **containerized** in a **Docker application**[1] that ensures portability and consistency. The application will feature a web interface for immediate and simple use, as well as an **API** that allows for programmatic extraction of **JSON**, enabling simple integration with other applications.

The Growing Importance and Challenges of Large Language Models.

Today, large language models are having a significant impact on the economy and society. For the first time, we have a system that can **interpret natural language on a wide scale** and transform it into actions (executable code) or coherent text (emails or website content).These models, such as GPT-3, can be used for numerous tasks, including **text generation, summarization, and question answering**. LLMs can dramatically improve productivity across different domains: they can enhance educational experiences, summarize complex legal documents, and expedite the extraction of critical information from judicial cases. In software development, LLMs can save substantial time in creating **functional applications**. Additionally, they serve creative purposes, such as writing stories, screenplays, and poems, and can augment existing datasets from tabular or text data. Given that text is the primary medium for human-machine interaction, the potential applications for the automatic creation of coherent text are truly limitless.

However, creating, running, and fine-tuning LLMs present significant **challenges**. Training these models requires **expensive computational resources**, making the process long and costly. Typically, only large corporations with substantial resources can afford it. As technology advances, increasingly larger models are being developed, but these open-source models, although pre-trained, are often **not fine-tuned for specific tasks**. Fine-tuning them often requires specialized, costly hardware, such as machines with extensive RAM and high-end GPUs. The latest NVIDIA GPU[2] models can range from at least $3000 upward. This price range is prohibitive for individuals or small entities aiming to fine-tune and run these models on consumer hardware. **Cloud computing** can alleviate this as it offers more accessible options, however renting virtual machines with attached GPUs can also become expensive. This is particularly true when fine tuning using the large datasets, extended training times can ramp up the cost.

---

[1] A containerized application can run efficiently anywhere as it is packed with all necessary code in one single “image”. Making it simpler to move from testing to deployment.
[2] GPUs are fast parallel processing devices, originally used for gaming and today fundamental for running LLMs.

(Efficient) Fine Tuning is All You Need.

When considering **large language models** in the current form such as the GPT (Generative Pretrained Transformers), we find that pretrained means that they already come with a certain volume of training and knowledge. Even the most basic models like GPT 2 have a good understanding of natural language, this means that the model is actually able to interpret human language and follow instructions. However, they often lack general or specific knowledge. The models are normally trained with large amounts of text, but not every single topic can be possibly covered.

To overcome this we can take the **pretrained models** that already contain a significant amount of training (eliminating the need for us to start from scratch) and use **transfer learning techniques** to fine tune with an additional task. The concept of transfer learning has its roots in psychology. A person who has studied a certain task, can transfer this knowledge, learned in one context, to another context.

In the context of LLMs, it is possible to say that the knowledge applied to one problem (answer generic questions) can be transferred to a new task (extract information) with ease. This is a type of **homogeneous transfer learning**, where the **feature space remains the same in both the original and the fine-tuned models**. In this case, the feature space consists of the words (or their numeric representations) in the text.

However the classic methods would require to fine-tune the entire feature space with all its dimensions. If this is the case, the practitioner would expect to face long training times and associated costs. This is because the transformers are very large neural networks with a substantial amount of parameters. On the other hand, **parameters efficient techniques come to help and solve this problem**. Often referred to as PEFT these are a series of techniques that aim to simplify or reduce the complexity of the feature space. By **decreasing the dimensionality of the attention** (Queries, Keys, Values, or matrices that focus the context) in LLMs we can also reduce the time that takes to fine tune them. **Leading to faster prototyping and cheaper development of useful models.**

The Hardware Needed For PEFT Fine Tuning.

In 2024 we will see new large language models appearing almost daily, with large corporations releasing new open source all the time. For example Llama 2 from Meta, and Gemini from Google. From 2 billion parameters, to up to 70 billion parameters. However, the problem is that even the smallest one could be several gigabytes of size and often the model needs to be loaded in memory all at once, therefore we need capable hardware.
The computational requirements are massive, and **GPUs** play a very important role. Originally these are video cards for gaming but since they can do parallel computations, they are very useful for running and training the transformer models in a timely manner.

It is worth noting that with the previous version of the attention mechanism it was not possible to do parallel computation. In fact the **Bahdanau Attention**[3] had a sequential dependency. The decoder generates each hidden state st based on the previous hidden state $st-1$.

$$e_{tj} = v_a^T \tanh(W_a s_{t-1} + U_a h_j)$$

Bahdanau attention computes the alignment score $\mathbf{e_{tj}}$, that requires the term $\mathbf{W_a s_{t-1}}$ which represents a matrix of weights applied to the decoder's hidden state from the **previous time step**. Because of this dependency the computation has to be completed sequentially, rendering virtually impossible parallelization.

On the other hand, the current version of the attention mechanism computes several attention heads independently, each one representing a certain linear transformation of the input sequence. Each head can be computed independently from the others. This allows the use of GPUs for faster parallel computation that in turn can process longer sequences.
In a nutshell the attention mechanism created by A. Vaswani and its team has been truly a game changer because of this redesigned computation. This allowed multiple processes to complete at once without sequential dependencies.

Some GPUs that can be used for loading and manipulating the models are currently the **A100** ($11000), **L4** ($4000) and **T4** GPU ($3000). These costs do not account for the rest of hardware such as powerful CPU and high amounts of RAM and electricity. It is easy to understand that for small entities these are important costs that need to be evaluated carefully.
Luckily cloud computing comes in handy. For fine tuning and inference, we can leverage rentable hardware and pay by the hour. In this case the cost will range from $0.40 to $10 per hour. Companies like **Hugging Face** offer **Spaces** that are containerized applications with attached GPU that can be automatically turned off when not in use. This is especially useful and cost saving for inference. Let's say that a company created an application based on a LLM, this software will continuously need the hardware even if no one is interacting with it. Hugging Face Spaces solution automatically turns off after a certain amount of time, and it will turn back on when someone uses it. The tradeoff is that the user might need to wait longer for the application to restart and become fully operational.

---

[3] https://machinelearningmastery.com/the-bahdanau-attention-mechanism/

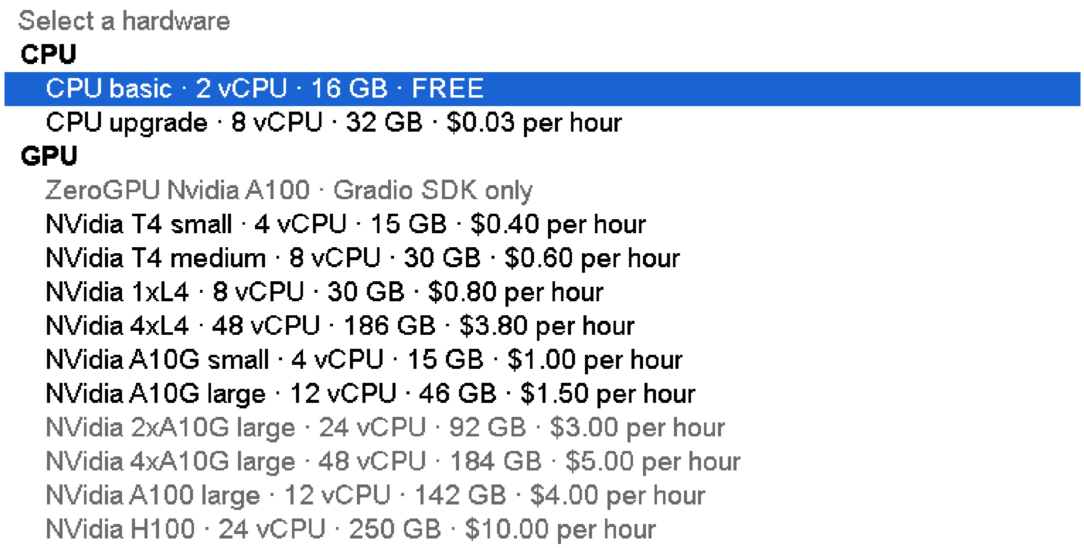
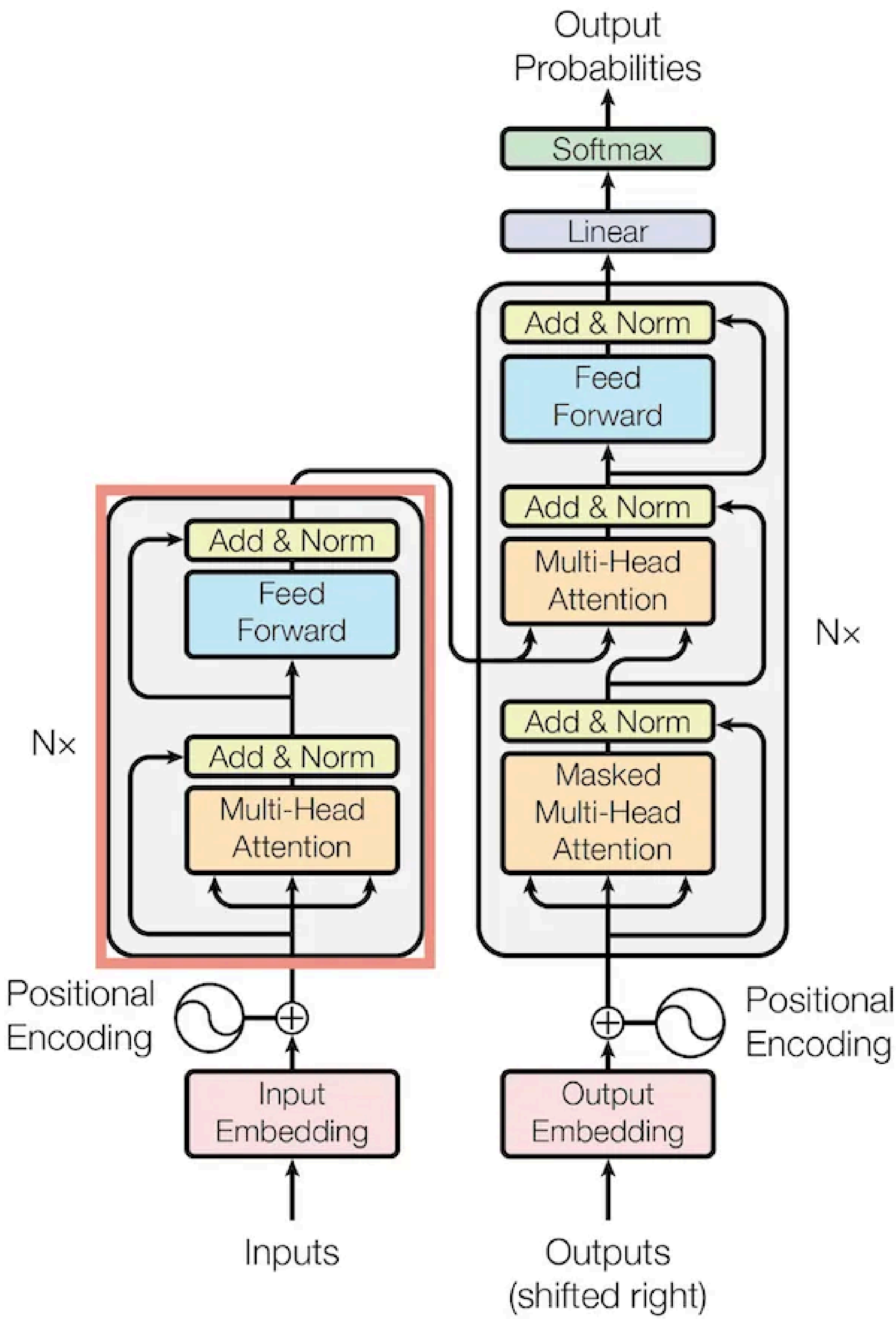


Fig. Hugging Face GPU solutions for GPU Inference and training LLMs

Chapter 2 Literature Review (Transformer Models and Fine Tuning Techniques).

The **aims** of this section are to discover the transformer architecture and understand it in fine detail exploring its foundational principles and the mathematical functioning. Therefore the **objectives** could be identified as **describing core components and analyzing the architecture**. With this objective in mind we can carefully describe with mathematical formulas and diagrams each part of its components. In other words, the following section will dissect its parts in detail, with explanations using an accessible language.
This groundwork will facilitate the exploration of fine-tuning techniques applied to the architecture to reduce its dimensionality. Consequently, the **aim** here is to discover and dissect **fine tuning techniques** that are applied to the architecture components to reduce its dimensionality. The **objective** is to find applicable practical solutions that can help to explain and implement these methods.

Dissection Of The Transformer Models.



The large language models that are dealt with in the research are based on the transformer architecture. This architecture appeared in 2017 for the first time in a paper named "**Attention Is All You Need**" by Vaswani et al. The paper argues that previous neural network architectures failed to “remember" long sequences of text, and they were slow, since the input was processed sequentially. Before the transformer architecture the state of art models were based on **Long Short-Term Memory Network** (**LSTM)** which are a type of recurrent neural network capable of maintaining some kind of memory. However this memory is normally lost when dealing with long sequences of text due to the **vanishing gradient problem**[4]. The transformer technology is a great improvement. The length of the sequences that are able to process are only limited by hardware.
Vaswani's discovery was that an attention mechanism was needed to help the architecture to focus on particular sections of the sentences, and the transformer architecture could be run in parallel (at the encoding stage), speeding up computation substantially, therefore the attention mechanism is able to capture all relationships for each word at once in parallel and not sequentially like the standard neural network. It must be also noted that this is not the only innovation, the transformer is in fact a complex neural network composed of several components.

If we look at the diagram, the transformer network is made of two main components. The one on the left is the **encoder** that takes fresh input. Its role is to process all tokens simultaneously and to understand their importance in the context. Thanks to the self attention mechanism, this encoded representation captures the meaning of the entire input.
The second part on the right is the **decoder** that uses the previous generated words (we can consider them as the context) and the **representation** from the encoder. We can see the decoder as the word generator. It is generated as a set of probabilities for the next word (logits). Transformers are autoregressive, in other words they predict the next word given a sequence, therefore the two parts together can capture meaningful relationships in long sequences of text and output the next most probable word.

“Guessing”[5] The Next Word.

The transformer works by “guessing” the next word in a sentence, often these models are referred to as **autoregressive models**, one word is added in front of the sentence, then the whole sentence is fed back in, and it continues like this guessing the next word. For example if your input is: “write a story…”, it will find the next most likely word, perhaps it will say “once” as the next word. Then it will start again and use the whole sentence again to generate the following word. Now the input is: “write a story. Once…”. The transformer will continue there adding one word each time.

[4] The gradient during backpropagation can become too small and vanish leading to stagnant learning.
[5] Guessing is intended as humorous play, as the model is actually predicting the next word based on a distribution of possibilities.

We all have used this in the past many times, maybe without realizing. For example, when you typed a message on your phone's email app, it suggested the next word for you. It is normally able to suggest several choices, based on the likelihood of the next word.

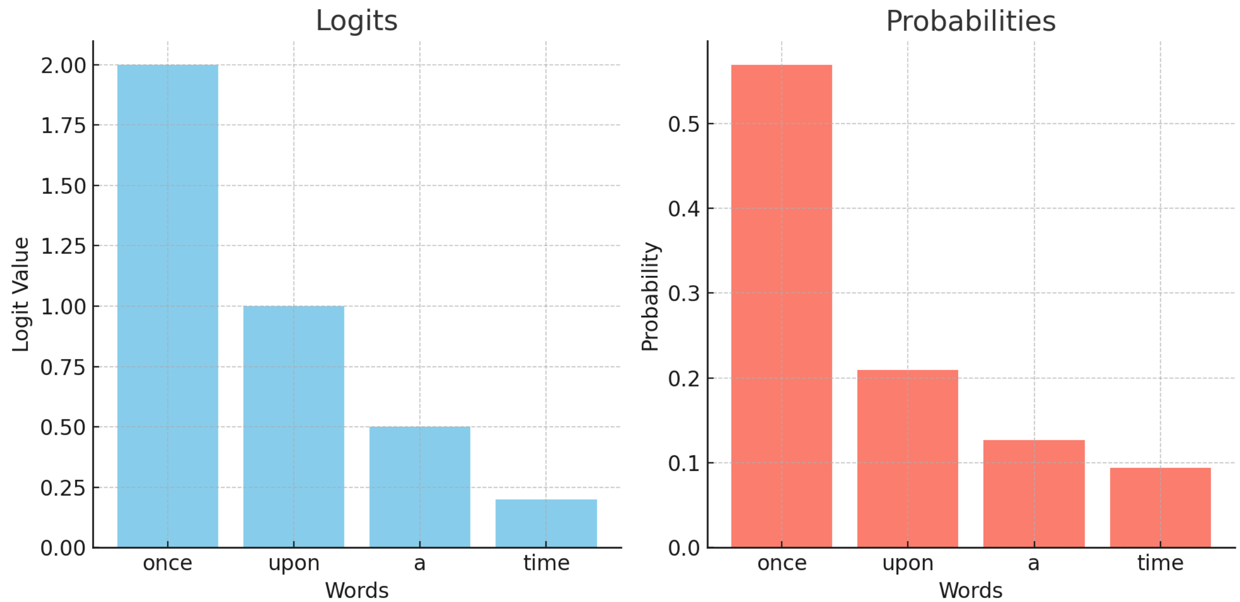


Fig. logit scores and their corresponding probabilities.

We can set up very simple examples where we receive the logits from the models. At this stage the logits are un-normalized word scores that were computed by the model. The last stage involves putting these results through the softmax function to obtain the probabilities. Above we can see that once is the highest probability in the sequence, followed by all the others.

The Encoder Layer.

As indicated earlier the transformer architecture includes two main parts: the **encoder** and the **decoder**. In this section we look at the encoder layer where new inputs are processed. Typically here we have preprocessing steps that create the embeddings, the self attention mechanism that is the heart of the model. Finally the computation goes through **normalization steps and a feedward network**, before passing the result to the decoder layer. The normalization step helps stabilize the learning process and the feedforward network captures more complex patterns beyond what the attention mechanism alone can provide. In the diagram we can also see that some of the connections are allowed to skip forward; this is to preserve some of the information from previous layers. The skip connections are part of the residual network, and this technique is used to preserve gradient information otherwise lost through the computation. Residual networks are important in preserving information and gradient flow throughout the network, making the training of deep models more feasible and effective.

Preprocessing Steps.

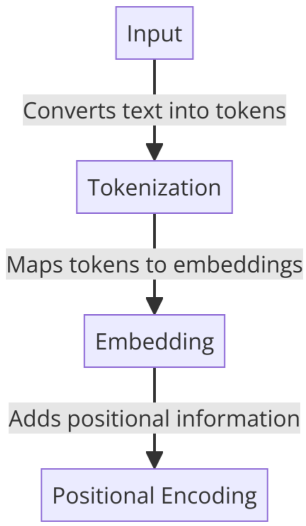


Fig. The preprocessing steps required in the transformer architecture

Data Preprocessing and Tokenization.

As computers do not understand words in the same way as humans do, at the very beginning of the computation the transformer creates a numeric representation of the words. This process is known as **tokenization**. First the text is broken down in single words. Depending on the method words are also divided into subparts. For example the word “doesn’t” is tokenized in “does” and “n’t”. This step is performed using predefined rules or algorithms. Each part of the text, including punctuation, is divided following the algorithm rules. Sometimes the start or end of the sentence too are assigned to a token too. This helps certain models to identify the beginning and end of text.

The tokenization process creates a vector data structure that is suitable for a machine learning model. **One example of preprocessing and tokenization is included in the appendix at the end of the document.**

Input Embeddings.

In the transformer preprocessing steps we also find the embeddings. This is part of the architecture that is able to learn and assign values to words based on their similarities. We can imagine the embeddings like cartesian coordinates (however in reality are high dimension vector spaces), and each word can be positioned on the space. Earlier versions of language processing architectures did not include the context of the words, therefore earlier systems were not able to make a difference between an **apple fruit** or an **Apple computer**. Later, we will see

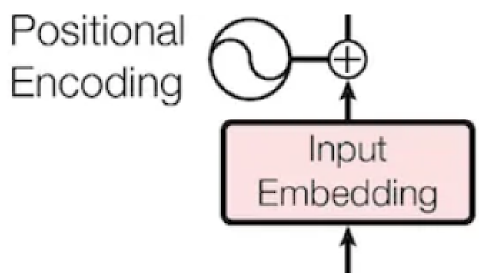
how modern transformer technology has included methods to understand the meaning of the words depending on the position in the sentence.
A simple visual example is that we can understand embeddings by imagining them as coordinates, we can imagine that words indicating **fruits** will be next to each other, but they will be far from words indicating **cars**.

In previous architectures predating the transformer, one common method to achieve embedding was to use word2vec, or word to vector method. In fact there were two main sub techniques in word2vec. The first is Continuous Bag of Words (CBOW) and the other is skip-gram. While the aim was the same, it was achieved with 2 different methods. In CBOW the aim is to guess a word given its context, while in skip-gram is the opposite or given the context to guess the word.

For example CBOW is a simple neural network that tries to estimate the matrix W weights given the input (a context window, normally a few word combinations near the target word) and the target outputs the word we are trying to predict. We start with a vocabulary that is one hot encoded and we create a context window C.

A simple example of one hot encoding is included in the appendix of the document.

Embeddings In The Transformer Technology.



The above example about previous architectures for embeddings is similarly applicable to the transformer technology however it also adds a further contextualisation by adding positional encoding to the embeddings.

Positional encoding takes care of **assigning an order to the words in the sentence**, this is not done semantically as a human would expect but by using mathematical sequences, such as the ***cosine functions***. In a nutshell, the functions take the position of the words and the total length of the vector space, and apply sine or cosine formulas depending on whether the position is odd or even. These sequences transform the position of the words in order to move them in the cartesian space and give them a unique position. For example if I have a sentence like "I like apples" and "apples like I", they contain the same words but one has a meaning and the other does not have a meaning at all. Without positional encoding they will have the same coordinates in the representation space. However with the function the coordinates are moved around and positioned differently. In a nutshell the function perturb their order. Eventually the 2 sentences will be positioned differently in the space, creating for each one a unique position, and therefore a unique encoding for each sequence.

If d is even:

$$PE(i,d) = \sin\left(\frac{i}{10000^{d/D}}\right)$$

If d is odd:

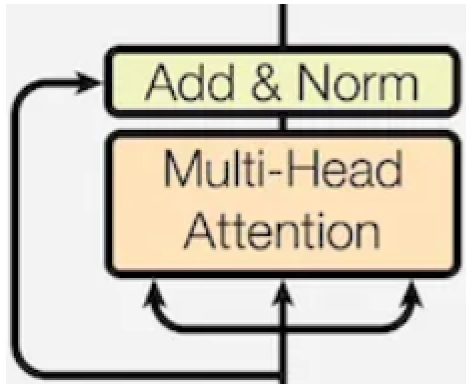
$$PE(i,d) = \cos\left(\frac{i}{10000^{(d-1)/D}}\right)$$

i: Index of the word in the sentence.
D: Total dimensions of each word embedding vector.
d: Specific dimension index within that embedding vector for which the positional encoding is being calculated.

In the above representation: i is the position of the word in the sentence. D is the total positions available in each word encoding so if the embeddings for one word are [0,4,5,2], D is 4, and d is the index of the position inside the vector in [0,4,5,2], therefore a number between 0 and 3.

In a nutshell, the formula alternates between sine and cosine based on whether **d** is even or odd. Therefore if a word has an **even position index** (inside the embedding vector), its sentence position **i** is divided by 1000 raised to the power of the ratio d/D. Finally the sine of the result is taken. This method ensures that the position of the word inside the sentence is effectively and variably encoded.

The Attention Mechanism.



In this section we focus on the attention mechanism which provides the core of the innovation in the transformer technology. In a nutshell we can define the attention mechanism as the **contextual relationships score** of each token (word or part of a word) when compared to the rest of the context. The attention enables the model to determine the significance of each word in a given text by computing how each token relates to every other token.

The Meaning From Context.

When we consider that attention mechanisms can distinguish words based on their context, the following example is helpful: we might have words that have different meanings depending on the sentence they are in, for example “seals are mammals”, and “this envelope has no seal”. In this case humans can extrapolate the meaning of the word from the context of the sentence. This ability is crucial because the attention mechanism discriminates meaning and associates the input with the correct prediction.

Imagine the tokens on a coordinate system, The word "**seal**" might be close to other **animals** or close to tools used for **sealing**. However since the word is ambiguous without context, perhaps it is sitting in the middle of the two categories. The attention mechanism allows these words to be pulled closer to one or the other categories depending on the context. Like a gravitational pull.

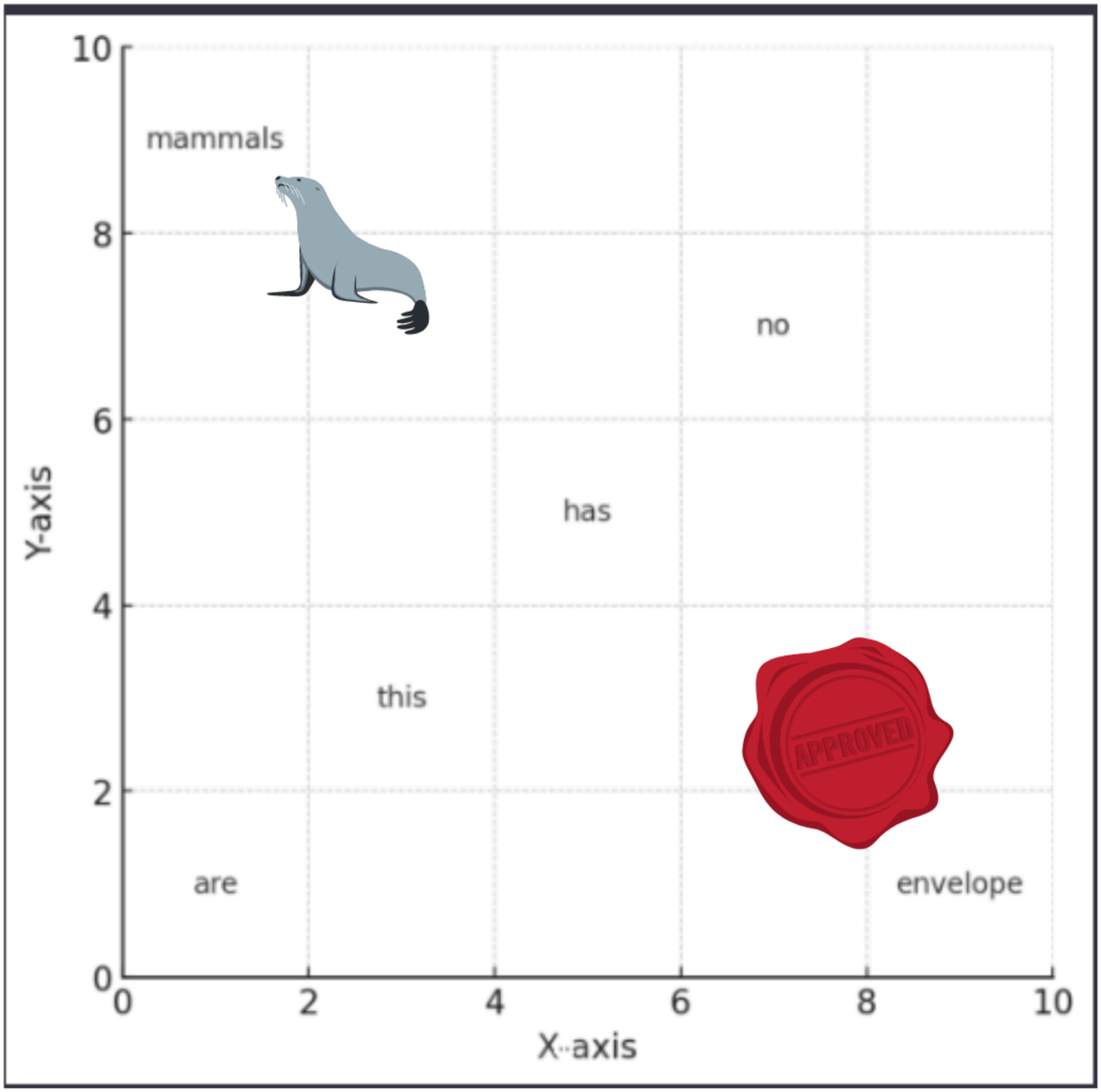


Figure: The two meanings of "seal" can be positioned differently in a conceptual space based on their usage in a sentence. "Seal" as an animal will appear closer to words like "mammal," while "seal" as a tool for sealing will appear closer to words like "envelope." Therefore the attention mechanism is able to differentiate between meaning depending on the words around it.

From the **preprocessing** we obtained the word encoding and the positional encoding, these two values are combined obtaining values that are **semantic and positional representations of the words in the sentence**. The attention mechanism in a nutshell uses a scaled dot product of the embeddings of the words. Each embedding is compared and moved around in order to get close to other words that **have similar meaning but this time using the whole context**. Therefore in the sentence “seals are mammals” the word seas will be moved closer to mammals, and in the sentence “this envelope has no seal”, the word seal will be moved closer to envelope. This helps to extrapolate the context of the sentence mathematically.

Queries and Keys. From Image Retrieval System to Attention Mechanism in Neural Networks.

The attention uses 3 learned through training matrices **K (keys), Q (queries) and V (values)**. This terminology is borrowed from the information retrieval system and it works in a similar fashion. Let’s imagine that you have a system to find images given a user search. The goal is to fetch relevant documents on a user's query. The user enters **Q or a query** that is the keyword for the search. The system maps the keyword to potential prelabeled images (title, and description) that are potential results. Finally the system scores the images according to what constitutes the best results or the best matched images. In the attention mechanism, V constitutes the output of the computation with the probability of the best results.

Similarly, in attention Keys and Queries are used to identify word-to-itself and other words' similarities through **linear transformations**. Like the image search engine example, the input is used to find titles and descriptions. V is a matrix of the original input's weights, giving context to each word. When these results are combined, we discover values that can be used to predict the next word from a vocabulary distribution. It is important to understand that Queries, Keys, and Values (QKV) are derived representations of the given word, or **spectral representations** of the word's relationships and context within the sequence.

For example if we examine **Q and K matrices** that are in fact linear transformations of the embedding. The transformations are able to demarcate the plane in a better way by “stretching or compressing the space”, this allows the attention mechanism to learn the complicated features that natural text has. The dot products represent the similarities of the words in the matrices. These Q and K matrices are found through a separate neural network that uses an optimisation process to learn the **best linear transformation**.

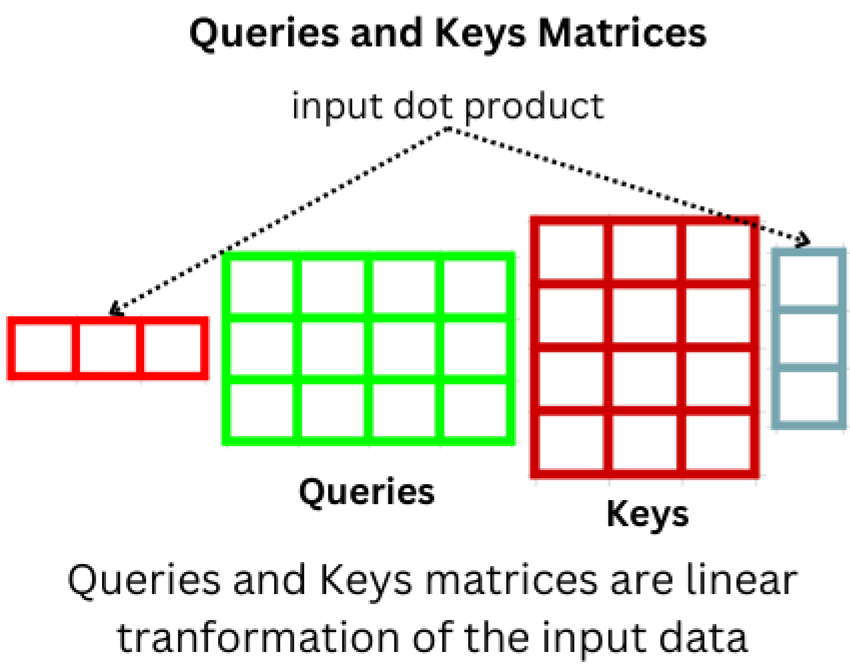


Fig.Queries and Keys matrices, derived from linear transformations of the text data, then the dot product measure word similarity.

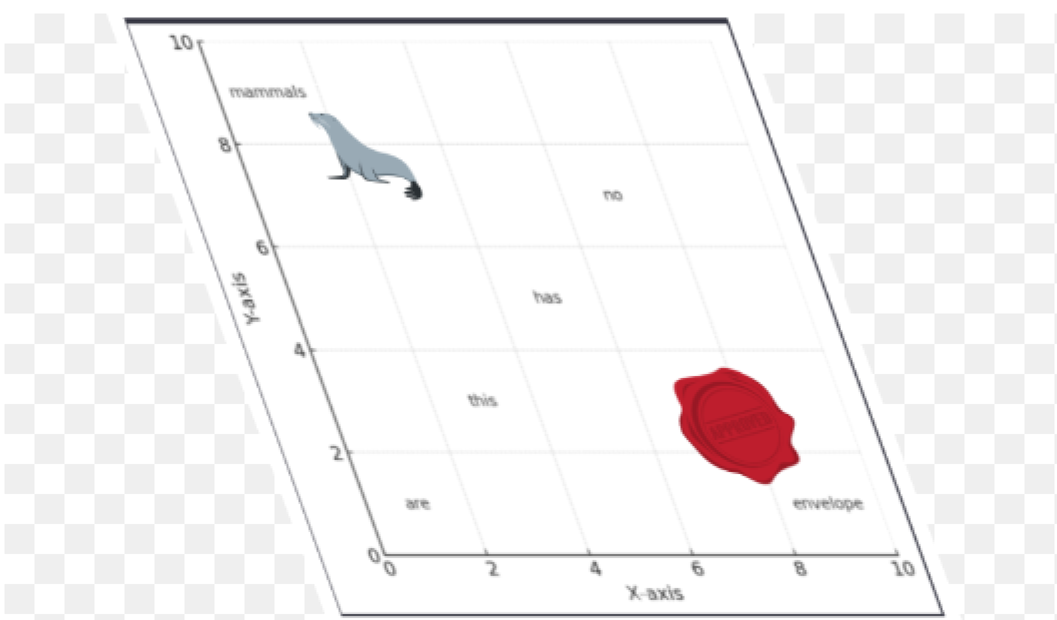


Fig. Linear transformations enhance the distinction between different word meanings based on context by skewing the space to extract features from the data.

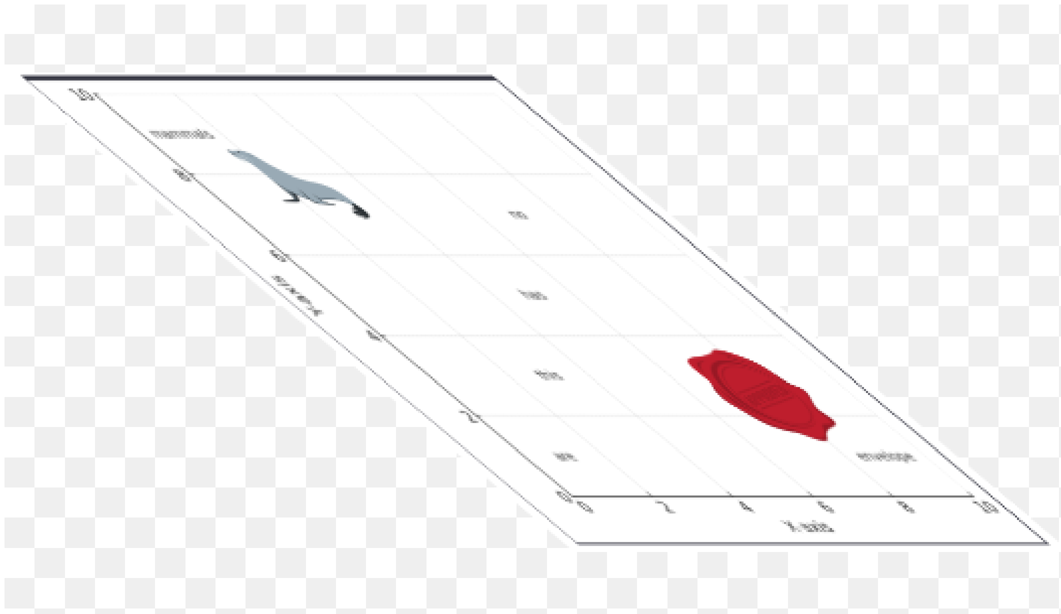


Fig. The attention mechanism finds a transformation that best represents the context. In this fictional representation, the two meanings of the word 'seals' are shown with increased distance between them, highlighting their distinct contexts.

The two fictional examples of the stretches and pulling are totally fictional. This serves to exemplify with images that words are pulled away from context that does not belong to them, while are attracted to relevant context.

Mathematical Explanation of Q and K.

Q,V,K are three matrices that are composed of weights **learned through training.** For a detailed explanation of their matrix representations and an example of how these matrices are structured, please refer to **Appendix.**

Softmax Function Comes Into Play.

After the dot product is calculated for Q and K, the output is passed through the scaling function to normalize the output and then transformed into probabilities via the softmax function .

$$\text{Softmax Scores} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

By computing the result of the dot product QK^t (transpose) and dividing it by the square root of the dimension of the K vector, we normalize the output. This normalized result is then passed through the softmax function, which computes the probabilities. **These results represent the contextual relationship as each probability indicates the relevance of the current word being processed.**

The Values Matrix.

Next we can start moving to the final steps of the computation. First we need to find the Value matrix. The matrix V represents a linear transformation learned during training, it is pretty much the same as for the Q and K matrices but again we need to find its weights W. When they are multiplied by the original embedding X, they result in V. The formula is as follows:

$$V = W^V \cdot X$$

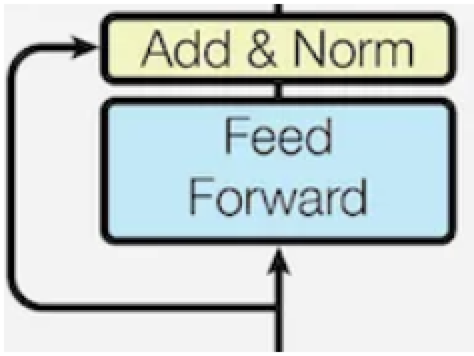
While Q and K are used for the attention scores, the V matrix contains the actual data. Therefore in the computation V provides a **weighted data representation**, fulfilling the role of **preserving information from the whole context.**

The Final Step Of The Attention Mechanism.

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

Now that we have previously calculated the softmax value, we finally multiply them by V, obtaining an aggregate value that represents the similarities of each word with itself and all the other words. With V multiplication we add values for the whole context of the sentence. The outcome of this multiplication is a comprehensive representation where each element is a blend of information from across the entire sentence, weighted by its contextual relevance.
With this word representation, an additional and final layer predicts the next word using the entire vocabulary probability distribution.

The Feedforward Network.



As we see in the previous chart, it is possible to notice that the large language model is a series of attention mechanisms **alternating with feedforward neural networks**. As computing the attention score introduces linearity in the model (when we did the dot product), we want to introduce non-linearity. In other words, the feedforward network **acts as a further improvement from the attention mechanism** enhancing the learning ability of the whole network. By introducing non-linearity we want to discover more complex patterns and high level relationships in the data. This is achieved using the ReLU (Rectified Linear Unit) activation function.

The ReLU is often used in neural networks as it is capable of creating and learning from nonlinear-transformations. It is the function of choice as it is very simple. Given an input it returns the same input if positive, or it returns zero it if the input is negative. The resulting matrix is the non-linear representation of the original data.

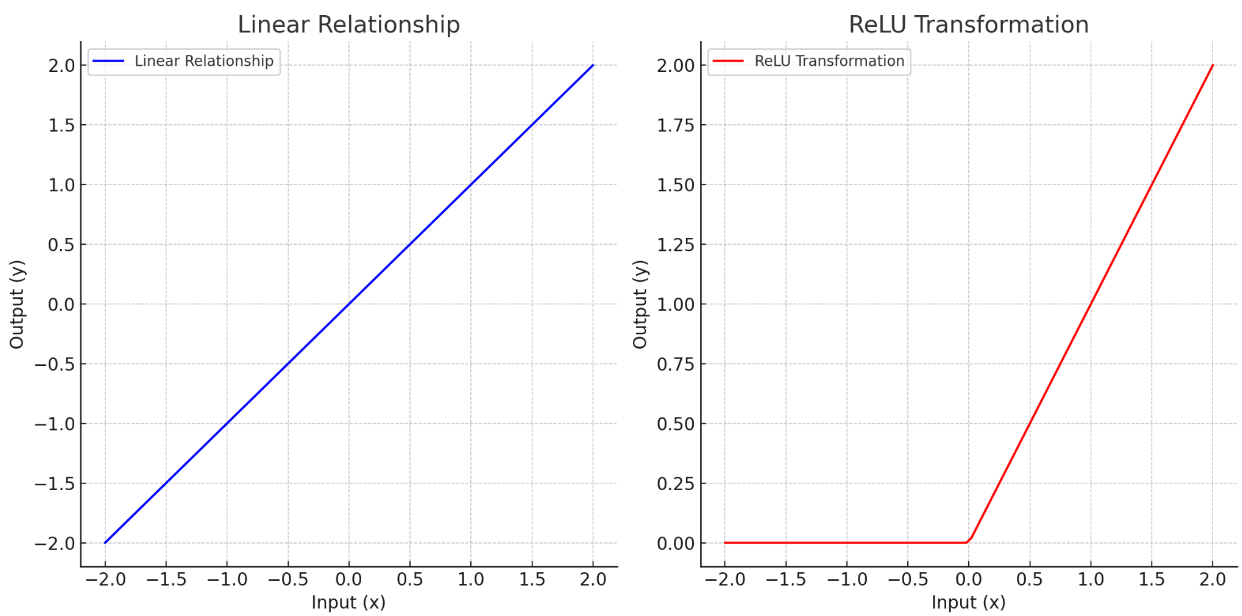


Fig. linear to nonlinear transformation applied in the feedforward network

As the name suggests, the data flows in only one direction: forward. The choice of this design lies in the simplicity of these networks, they are simpler to train, easier to debug and less prone to overfitting compared to other network types. Using the transformed input, this type of network will learn its own weight, and can eventually develop a high-level representation of the patterns in the data.

At each step of attention and forward network the data will reside inside this particular layer, and it will be a representation of that particular step. These layers are of not simple interpretation, however we could suppose that the early layers could be a representation of the basic semantic information of the words or simple syntactic structures. As we move forward to the middle layers, the model could start to identify sentences and clauses. Finally the last layers could capture the global relationship in the text. This process could be seen as hierarchical. The first layers have a more local focus and the top layers that have a more global understanding.

Residual Connection (Addition) and Normalization Step.

These 2 steps help the network to **learn better, and avoid some common problems**. Over the long computation some information is lost due to vanishing or exploding gradient issues. By "**skipping**" some of the connections this can be avoided. Normally referred to as ResNet. This part of the architecture is able to learn from previous gradients and bridging the information to new connections by preserving the information. This step helps in a better gradient flow, easier optimization, and provides flexibility in learning.

The concept involves summing the output of a layer with its input, expressed as:

$$y = F(x, \{W_i\}) + x$$

This idea was introduced by **Heet al. in their 2015 ResNet model**. It serves as one of the solutions to the vanishing gradient problem. Where the output of the network $F(x, \{W_i\})$ is added to the input x. This allows the information of the input to "skip" the transformation that happens inside the function therefore preserving the gradient.

Finally layer normalization helps to stabilize the activations and also prevent issues like exploding and vanishing gradients. Since deep neural networks can become overly complex, normalization improves convergence and helps maintain consistent activation distributions.

$$\mathrm{LayerNorm}(x) = \frac{x - \mu}{\sigma} \cdot \gamma + \beta$$

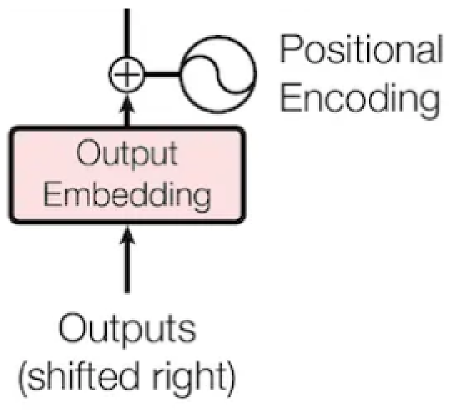
where μ and σ are the mean and standard deviation of the activations, and γ and β
β are learnable parameters that allow the normalized activations to be scaled and shifted.

Finally we put this together, the residual connection (Addition) and the layer normalization we obtain the following formula.

$$y = \text{LayerNorm}(x + F(x, \{W_i\}))$$

These 2 steps combined together help to avoid common problems of deep neural networks by ensuring stable gradient flow, reducing internal covariate shift, and facilitating the training of deeper networks without significant performance degradation.

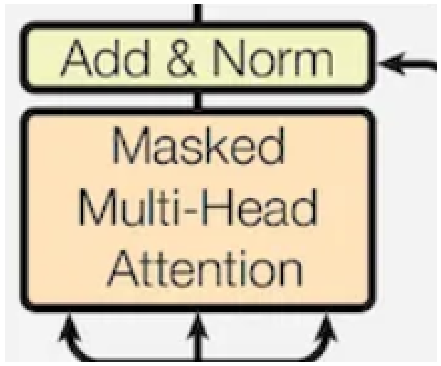
The Transformer Model Dissection Continued: The Decoder.



In the diagram, the right part of the transformer architecture is the **decoder**. This section will actually **output the probabilities** of the next word. We can think of this part as responsible for generating the sequence based on the input.
There are 2 distinct stages. It behaves differently depending on whether it is inference or training time.
At **inference time,** the main **input of the decoder are the previously generated embeddings.** At this stage they constitute **the context for the prediction of the next word** (in a sequence generation problem).
Otherwise, **at training time** the embeddings are obtained by the ground-truth target sequence. We train the model with the sequence in the original language and also present it with the translation in the target language. This is also called **teacher forcing**. We aim to show the model the answer to the problem. The model learns to represent and generalize the data.

As we continue with the diagram we see that some steps of the decoder are repeated (positional encoding) so let's focus on the parts that we haven't discussed so far.

Masked Multi Head Attention.



Earlier, we discussed the attention mechanism, which is at the core of this architecture. This step is repeated multiple times throughout the network, and we see it again here in the decoder. The transformer architecture uses **a special version of the attention: the multi-head attention**. In fact, it computes **several versions in parallel of Q,K,V matrices.** Each captures different transformations of the data. The parallelization of the operation that takes advantage of the GPU architecture offers faster computations. This is one of the key innovations of the attention mechanism in transformers

The formula below exemplifies the calculation where multiple copies of the computations are linearly projected h times.

$$Q_i = QW_i^Q, \quad K_i = KW_i^K, \quad V_i = VW_i^V \quad \text{for} \quad i = 1, 2, \ldots, h$$

This is followed by a concatenation

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)$$

Finally the concatenation is linearly transformed using a set of learned weight Wo

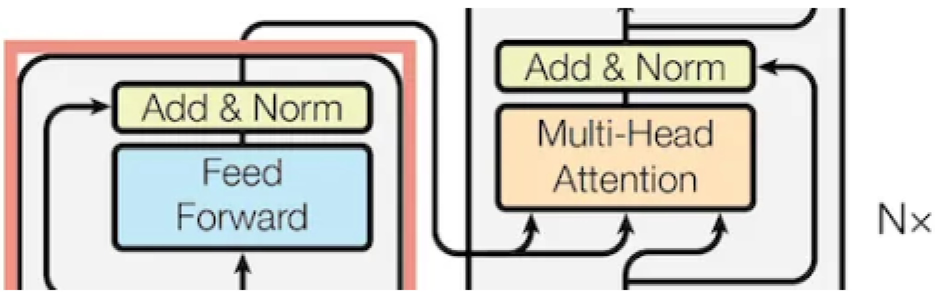
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O$$

During training time, when the decoder is presented with the input sequence and the ground-truth sequence, it also applies a mask to the sequences. This mask prevents the decoder from looking ahead and "cheating." Therefore, the decoder can only see the past tokens and the current token that it is trying to predict. The mask effectively conceals all the future tokens from the sequence. This is important as it emulates a real case scenario where it can see only the past context and it cannot rely on future tokens.

Flow From Decoder Output to Linear Layer and Softmax.

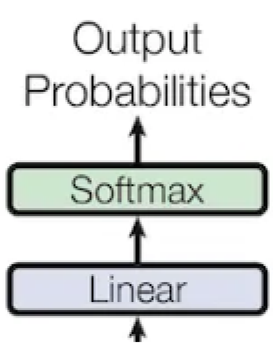
As the data flows through the computation, several steps are the same in both the encoder and the decoder. Now, we can focus on other aspects not previously covered: the flow of the fresh input from the encoder to the decoder and the final project through the linear layer. Finishing with the softmax activation function.

Insertion Of Data From Encoder To Decoder.



The fresh input from the encoder is added at this stage to the flow of the computation of the decoder that contained the previous input. As we can see in the diagram the data flows through another multihead attention and an Add & Norm layer.
By combining the tokens, the transformer effectively generates the correct sequence that will eventually lead to the prediction of the next token given the combination of the previous sequence and the new word.

Linear & Softmax



At this point the transformer model has produced a sequence of vectors. Each vector represents a token that appears in the text; the vector contains the model's understanding of the context in regard to the position in the sequence.
For example if the model was trained on “Puss in Boots" fairy tale, in respect of the word “cat”, I would have a high dimension vector like this

$$\text{vec}_{\text{cat}} = [0.1, -0.3, 0.7, \ldots, 0.4]$$

This would represent the context understanding in respect of the single word when compared to the whole story.

This vector is then passed to a linear transformation function indicated in the diagram.

$$\text{logits}_{\text{cat}} = \text{vec}_{\text{cat}} \times W + b$$

This step helps with **dimensionality reduction by mapping the output to the size of the vocabulary in the text**. The mapping effectively transforms the high-dimensional vector, which contains the model's understanding, into a vector of the same size as the vocabulary, preparing it for the softmax activation function without significant loss of important information. Moreover, linear transformations combine features learned in previous layers, integrating all outputs into a single transformation.

Finally the **softmax function** receives raw unnormalised output (logits) of the model confidence of the potential next token. The softmax function very simply transforms the data into probabilities. After softmax, the whole vector would add to 1. The final step is to take the highest probability form the vector. This would be the next token in the sequence.

For example if our logit has 3 word vocabulary it could look this this

$$\text{logits} = \left[-0.3, 0.33, 0.71\right]$$

Then the softmax function taken them in:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

Where each logit is exponentiated transforming them into a positive value. Followed by a normalization step The exponentiated values are then divided by the sum of all exponentiated values. This ensures that the result will be a value between 0 and 1 effectively being a probability.

The output of the softmax function could be something like this:

$$\text{probabilities} = [0.178, 0.334, 0.488]$$

Each value is the probability of the next token for the output. At this stage the model picks up the highest value of 0.488 as the potential final output. This concludes the computation and the prediction.

Further Thoughts About The Attention Mechanism with Code.

The full colab example is available here
https://colab.research.google.com/drive/1hNKEQSqBhKqcx2MTiRl-b4qs8dVfgSAQ?usp=sharing

Let's start with the sentence "seals are mammals", these are example endeddings that represent the words in space. We can see that the words seals and animals are close to each other.
With this example we want to see how the attention mechanism focuses on a particular word.

```
embeddings = {
    "seals": [0.7, 0.1, 0.7],
    "are": [0.0, 1.0, 0.1],
    "mammals": [0.8, 0.1, 0.7]
}
```

At this stage we need to compute 3 weight matrices that represent Q,K,V. These are learned weight matrices computed when the word "seal" is compared with itself and the rest of the sentence.
"Seal" is our input and focus word. "Seals are mammals" is our desired output, therefore we set up a simple network to learn the weights.

```
embeddings = {
    "seals": [0.7, 0.1, 0.7],
    "are": [0.0, 1.0, 0.1],
    "mammals": [0.8, 0.1, 0.7]
}
# Prepare input and output data
input_embedding = np.array([embeddings["seals"]])
output_embeddings = np.array([embeddings['seals'], embeddings["are"],
embeddings["mammals"]])
# Define the neural network
model = Sequential()
```

```
model.add(Dense(3, input_shape=(3,), activation='linear'))
model.add(Dense(3, input_shape=(3,), activation='linear'))
# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')
# Train the model
model.fit(np.tile(input_embedding, (3, 1)), output_embeddings,
epochs=1000, verbose=0)
# Get the learned transformation matrix
W_Q= model.layers[0].get_weights()[0]
```

The output of the computation is below. The output represents the learned matrix through training. It is the linear transformation of the word "seal" when compared to the rest of the sentence.

W_Q=[[-0.9639853 -0.370478 0.03261321]
[0.7767219 -1.0498722 -0.46989882]
[0.5529736 0.23820241 -0.14223664]]

K_Q=[[0.40419954 -0.2877785 0.05992314]
[-0.86603606 -0.62761724 -0.5118438]
[-0.70111537 -0.39987478 -0.3694805]]

V_Q=[[-0.66045386 -0.9967126 0.32280177]
[0.7664507 0.49089956 -0.6917728]
[0.49353847 0.21592456 0.658109]]

Then we process the weights through the attention as follows:[6]

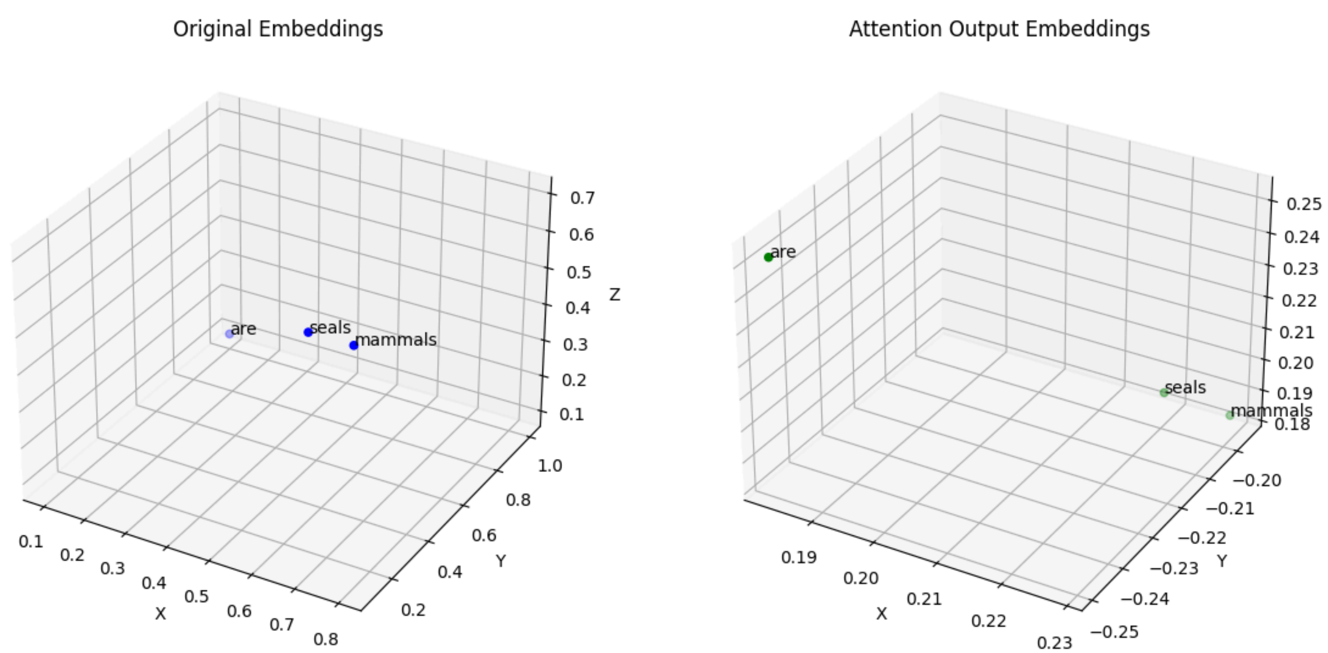
```
# Encoder representations of four different words
word_1 = np.array([0.7, 0.1, 0.7])
word_2 = np.array([0.1, 1.0, 0.1])
word_3 = np.array([0.8, 0.1, 0.7])
#word_4 = np.array([0, 0, 1])

# Stacking the word embeddings into a single array
words = np.array([word_1, word_2, word_3])

# Generating the queries, keys and values
Q = words @ W_Q
K = words @ W_K
```

[6] Example code from https://machinelearningmastery.com/the-attention-mechanism-from-scratch/

```
V = words @ W_V

# Scoring the query vectors against all key vectors
scores = Q @ K.transpose()

# Computing the weights by a softmax operation
weights = softmax(scores / K.shape[1] ** 0.5, axis=1)

# Computing the attention by a weighted sum of the value vectors

attention = weights @ V
```

attention result:
[0.22177987, -0.20313328, 0.1936999],
[0.18234888, -0.24927167, 0.25210221],
 [0.22883756, -0.1948938 , 0.1831917]

Then we can proceed to plot the result as follows:



On the left, we have the original embeddings. We see that there is a similarity between “seal” and “mammals”. However when we compute the attention in reference to the word “seal” and the rest of the sentence, we see that the word “are” has been pushed away. With this simple example, it clarifies how the attention mechanism “attend” or focus on certain parts of the text.

What Role Does The Attention Has in Lightweight Fine Tuning?

In the regards of **lightweight finetuning,** we want to target one main objective. We have to identify layers of the language model that are suitable for this operation. In most cases the attention matrices Q (query), K (key), and V (value) will be the target as they are high dimensionality representation of the linear transformation of the text. Often these matrices can be compressed to lower dimensions without much loss of information. The compression directly translates to a lower computational load and consequently a reduction in cost to produce task specific models. In other words we want to preserve the complex representation of the main model and pass its capabilities to a more specific model that has been trained to complete our task.
Secondly, it is important to identify techniques that can actually implement the fine tuning. In this research we will explore 5 techniques such as **LoRA (Hu et al., 2021) Low-Rank Adaptation. Then we will look into Adapter Modules (Houlsby et al., 2019), followed by Adapter Modules (Houlsby et al., 2019). Finally Prefix-Tuning (Li and Liang, 2021), Llama-Adapter (Zhang et al., 2023), and Intrinsic Dimensionality (Aghajanyan et al., 2020).**

How LoRa Reduces the Complexity of Large Language Models.

LoRa stands for Low-Rank Adaptation. We can fine tune LLMs without the need of expensive and extensive resources. This method involves finding a layer that can be reduced in its complexity while maintaining a similar output, in most cases **we will target only the attention layers and freeze the rest**. This allows us to update only a smaller set of parameters.

This is achieved by finding **2 smaller low ranking matrices A and B**, which **approximate the weight matrix W**. Some details will be lost, however the method has proven to produce good results. A and B are low ranking rectangular matrices that when multiplied together will result in the original weight matrix that we are trying to compress. In a nutshell low ranking matrices are matrices that have fewer linearly independent column vectors than the maximum possible for their dimension, therefore the data representation in them is less complex. By finding A and B and then multiplying them together to obtain W, the computation is much less expensive than using the full ranking W matrix.

The math behind it, assumes that there is W_0 which is a n x n pretrained dense layer. Then we have matrix A, which is n x rank, and another matrix B, which is rank x n in shape. In the original paper, the hyperparameter “rank 4” is used and it is much smaller than n[7]. This means that the matrix product A x B significantly reduces the number of parameters compared to a full n x n matrix.

The original equation is:

$$output = W0x + b0$$

Where x is the input and W0 the weight matrix and b0 is the bias term.

The LoRa modified equation is as following:

$$output = W0x + b0 + BAx$$

Where A and B are the rank decomposed matrices.

The point behind this approach is that large language models are over parameterized. We need to find the layers that have a low intrinsic dimension. Therefore, by finding A and B we hope to reduce the intrinsic dimension of the model, compressing the valuable and important data into more meaningful matrices.

Experimenting with LoRa and Keras in Colab.

https://colab.research.google.com/drive/1nfTV_kBEdSTBC9reUW_RPoZ_cDIenTTh?usp=sharing

This Colab experiment[8] is based on the ideas from the notebook "Parameter-efficient fine-tuning of GPT-2 with LoRA" by Abheesht Sharma and Matthew Watson, available on KerasNLP's GitHub repository[9]. This experiment aims to demonstrate the ability of fine tuning and transfer learning.
First, the standard GPT-2 model was queried with the classic Shakespear quote “To be, or not to be, that is the question". **The model's answer was disappointing and unrelated to the question.** This happens because the model had not been exposed to this type of text before, resulting in an incoherent response

---

[7] Source https://keras.io/examples/nlp/parameter_efficient_finetuning_of_gpt2_with_lora/
[8] LoRa_shakespeare_experiment
https://colab.research.google.com/drive/1nfTV_kBEdSTBC9reUW_RPoZ_cDIenTTh?usp=sharing
[9] Parameter-Efficient Transfer Learning for NLP
https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/nlp/ipynb/parameter_efficient_finetuning_of_gpt2_with_lora.ipynb

```
Generated Text:
1: To be, or not to be, that is the question.

The question is, what is it?

The question is, what is it?

The question is, what is it?

The question is, what
```

Fig. the text when prompted with “To be, or not to be, that is the question"

In the experiment, the next step was to fine-tune GPT-2 with the Tiny Shakespeare dataset to improve its performance when asked Shakespearean-style questions. In the first attempt, no modifications were made, and the model contained 124,439,808 parameters, all of which were trainable.

```
Total params: 124,439,808 (474.70 MB)
Trainable params: 124,439,808 (474.70 MB)
Non-trainable params: 0 (0.00 B)
```

Fig. The parameters from the original model.

Experiment finding:

The model was retrained using the LoRa technique, and these are the main findings.
The original GPT-2 model training took 121.15 seconds. The LoRA-enhanced model trained significantly faster in only 63.7 seconds.There was an improvement also during inference. The original model took 15.99 seconds to output the answer, while the LoRa model took only 10.63 seconds. Moreover, both models had a notable and similar improvement in coherency.
This is due to the application of the LoRa reparametrization, as the attention matrices were reduced, while the rest of the model remains frozen. As a consequence, the trainable parameters are substantially reduced to only 147465. This leads to a faster performance while maintaining a similar quality of coherency.

```
Total params: 124,587,264 (475.26 MB)
Trainable params: 147,456 (576.00 KB)
Non-trainable params: 124,439,808 (474.70 MB)
```

Fig. the parameters of the LoRa model.

The output of the LoRa modification maintains a high level of coherency and it demonstrates that it understands the question in a philosophical way.

```
Output:
To be, or not to be, that is the question.
I'm not a lawyer, and I'm certainly not a
lawyer who wants to be a lawyer.

But what I am doing is trying to make a
difference for the future of our country.
I've been in politics for 15 years and I've
been elected by a majority of the people in
our state, but I've been in the state Senate
for 15 years, and it's not something I've been
able to do in a long time. I've got to do it
now and I think it will be worth every minute
of my time.
```

Fig. the output from the LoRa model

The LoRA-enhanced model used slightly more GPU memory. This was not expected, the original Colab memory usage was more efficient. In our case this does not happen and it might be ground for further investigation. **However the key finding is that LoRA enables efficient fine-tuning with faster training times while enhanced text generation capabilities are maintained**.

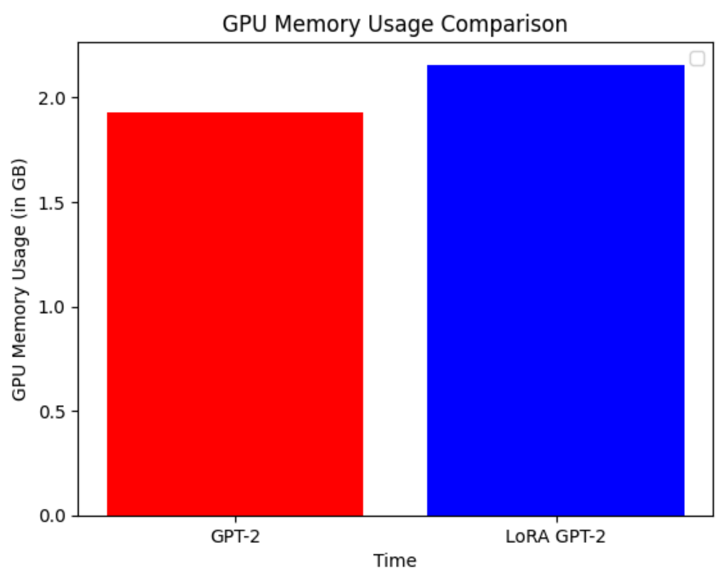


Fig. Memory usage during training

Adapter Modules.

Adapter module is a technique for lightweight fine tuning mentioned in the paper "Parameter-Efficient Transfer Learning for NLP" first published in 2019 (Houlsby et al., 2019). https://arxiv.org/pdf/1902.00751.pdf

This is another method that considerably reduces the resources needed for fine tuning. In the aforementioned paper, this method demonstrated to maintain a very high level of accuracy **while updating only a small amount of parameters via a trainable adapter**.
Adapter modules are small neural networks that are inserted between the major layers of the transformer models. When this technique is applied, the original layers are frozen (**weights are not updated**), and we will **update only the adapter module.**

Another potential implication is that by maintaining the same backbone, it is possible to fine tune the model for **different tasks**. This makes this method particularly interesting. We could have a model that can efficiently do translations, question answering, and summarization. In a nutshell, it is possible to **integrate multiple adapters** into the same LLM, allowing for versatile performance across various applications.

In the adapter model fine-tuning, we can achieve feature transfer by considering the original matrix, where ϕ is the original function used to learn the weight based on the input x, as per the following formula:

$$\phi_w(x)$$

The aim of the adapter is to discover a new function that can approximate ϕ. Therefore, if we can find a new function Ψ, it can be used to update only a small amount of task specific parameters indicated by v.

$$\psi_{w,v}(x) \approx \varphi_w(x)$$

The adapter module incorporates a down projection (bottleneck), achieved through a linear transformation that **reduces dimensionality of the data**; this method allows the network to learn complex task specific features. Finally, the network **projects the data upward, returning** it to its original dimension. During these steps, the dimensionality of the data is reduced, resulting in an approximation of the original weight matrix. **This down projection followed by upward projection is substantially faster than working with the original matrix.**

For the adapter task two Colabs were created.

1) **https://colab.research.google.com/drive/1aPZpHEozsJ-wB8_3bgiYqIds8Bj4cJzf?usp=sharing**

The first Colab is for sentiment analysis and it based on the original code provide by Adapter-Hub[10] which performs very well on unseen data. The code leverages the adapter module from the transformers library. It applies a bottleneck adapter at the output level of the RoBERTa (Robustly optimized BERT approach) model. The adapter is trained on a new task for sentiment analysis using the IMDB data. This adapter training is integrated before the model's output layer. In the Colab, we can see that it works very well at classifying the sentences with their correct sentiment score.

2) **https://colab.research.google.com/drive/1GmLtXa53DO1niSOVnNHVAxfN5S_gaYOq?usp=sharing**

The second Colab attempts a more complex task but fails. This time, a summarisation adapter is created and attached to BART (Bidirectional and Auto-Regressive Transformers), however it never seems to learn the task. It outputs either the noise in the text (probable overfitting) or not learning the task at all by outputting the same text (underfitting).

With the **weight watcher library,** it is possible to compare the two models' weight distribution (a table is shown below). The summarization model shows higher overall magnitude, greater complexity (as indicated by Stable Rank), and increased stability (Log Spectral Norm), yet it does not perform well at all at the inference stage despite good results in training and evaluation loss metrics.
The sentiment analysis model demonstrates a **more pronounced heavy-tailed behavior** (higher alpha value), this means that it is more long tailed than the summarization model, indicating it is better at capturing the variance of the data. Also, the sentiment analysis model has a lower stable rank indicating a more concentrated distribution of values. This means that fewer dimensions can capture most of the information. Leading to a better performance overall.

That being said, we also have to consider that the sentiment analysis task is much simpler than summarization. The latter requires further investigation, as the library itself might not fully support the task with the model used[11].

---

10 https://colab.research.google.com/github/Adapter-Hub/adapters/blob/main/notebooks/01_Adapter_Training.ipynb

11 What method can be used for the available models https://huggingface.co/docs/peft/index

Comparison Table with Reference Values

Metric	Summarization (First Set)	Sentiment Analysis (Second Set)	Reference Values
Log Norm	3.46	2.86	2.5 - 3.5
Alpha	4.59	5.99	3.0 - 5.0
Alpha Weighted	7.68	5.12	5.0 - 7.0
Log Alpha Norm	8.07	5.41	5.0 - 7.0
Log Spectral Norm	1.79	1.33	1.0 - 2.0
Stable Rank	56.20	40.14	30 - 60

Fig. The comparison of the 2 models from the weight watcher library.

Prefix-Tuning.

Experiments have been conducted in Colab for the prefix fine tuning method and the notebook can be found here
https://colab.research.google.com/drive/19WSr0WWqZ52TAjgtRPKvxjMWiV1zVv_d?usp=sharing

This method was proposed in 2021 by the paper "Prefix-Tuning: Optimizing Continuous Prompts for Generation" by Xiang Lisa Li and Percy Liang from Stanford University. The paper is available at https://arxiv.org/pdf/2101.00190.pdf. Like all the methods we have seen so far, the aim is to freeze the original model weight matrices and update only a small amount of parameters. In this case, we create a small set of task specific parameters called “prefix”. The prefix is a learned matrix of parameters prepended to the prompt. This method allows the model to incorporate task specific knowledge.

Since these models are autoregressive, each output is influenced by the previous one, meaning the model generates output one token at a time. Therefore, by pre-inserting a prefix containing fine-tuning information, we can influence the output without the need to retrain the whole model. Let’s imagine we want to fine tune a model using prefix tuning, first we need to generate P.

$$P = \{P_1, P_2, \ldots, P_n\}$$

P is a sequence learned during training. It is the same space of the hidden states (in terms of dimensionality and structure for seamless integration) of the original model. Quoting from the original paper “Prefix-tuning initializes a trainable matrix Pθ (parametrized by θ) of dimension |Pidx| × dim(hi) to store the prefix parameters.", where |Pidx| represents the length of the prefix sequence and dim(hi)refers to the dimensionality of the hidden states.

P represents the fine tuning data that will be **prepended** to the model.

$$[P; x] = \{P_1, P_2, \ldots, P_n, x_1, x_2, \ldots, x_m\}$$

The objective is to obtain this modified sequence where P is our prepended fine tuning data, and x is the standard prompt sequence.

$$h_i = \text{Transformer}([P; x]_{<i}, h_{<i})$$

The formula represents how the data is processed by the transformer. While the prefix tokens are not directly processed by the transformer model itself, they influence the sequence x by leveraging the hidden states already present in the model. **This means that the prefix provides additional context that guides the generation process**. It helps it produce more accurate and task-specific outputs.

Given a position in the hidden states i, [P;x] < considers only the concatenated elements before the current position i, while h <i considers only the elements before the hidden state i. This represents the elements that have been influenced in the hidden states by the modified sequence

Finally, the ***hi* = Transformer()** represents the computation in the transformer of both the concatenated elements and the hidden states. To clarify, the prefix modification aims to **influence** the existing hidden states without directly modifying them. Hence, the concatenation does not modify the hidden states. The prefix provides additional context, thereby **influencing** the computation of the hidden states **indirectly**. Quoting from the paper “The fine-tuning... optimizes a small continuous task-specific vector (called the prefix). Prefix-tuning draws inspiration from prompting, allowing subsequent tokens to attend to this prefix as if it were “virtual tokens…”

$$\max_{\theta} \log p(y | [P; x]; \theta)$$

The final equation of the prefix method represents the objective that is to maximize the log probability of generating a sequence y given the modified output P and x using the modified model theta parameter.

Summarization Task Using Model T5 and Prefix Tuning.

In the attached Colab, the summarization task using the CNN/DailyMail dataset and the T5 model was attempted again. This time, the model successfully summarized the given information.

```
text="Former U.S. President Donald Trump was shot at during a rally in
Butler, Pennsylvania, but is now doing well despite being hit in the ear.
The attack resulted in one bystander's death and critical injuries to two
others. The shooter, identified as 20-year-old Thomas Matthew Crooks, was
killed by Secret Service at the scene. The incident is being investigated
by the FBI, Secret Service, and the Department of Homeland Security, and
it's treated as an assassination attempt. Secret Service Director Kimberly
Cheatle will testify before the US House Oversight Committee regarding the
incident. President Joe Biden and other political figures have condemned
the violence."
```

And the output is:

```
['Trump was shot at during a rally in Pennsylvania. One bystander was
killed and two others critically injured. The incident is being
investigated as an assassination attempt.']
```

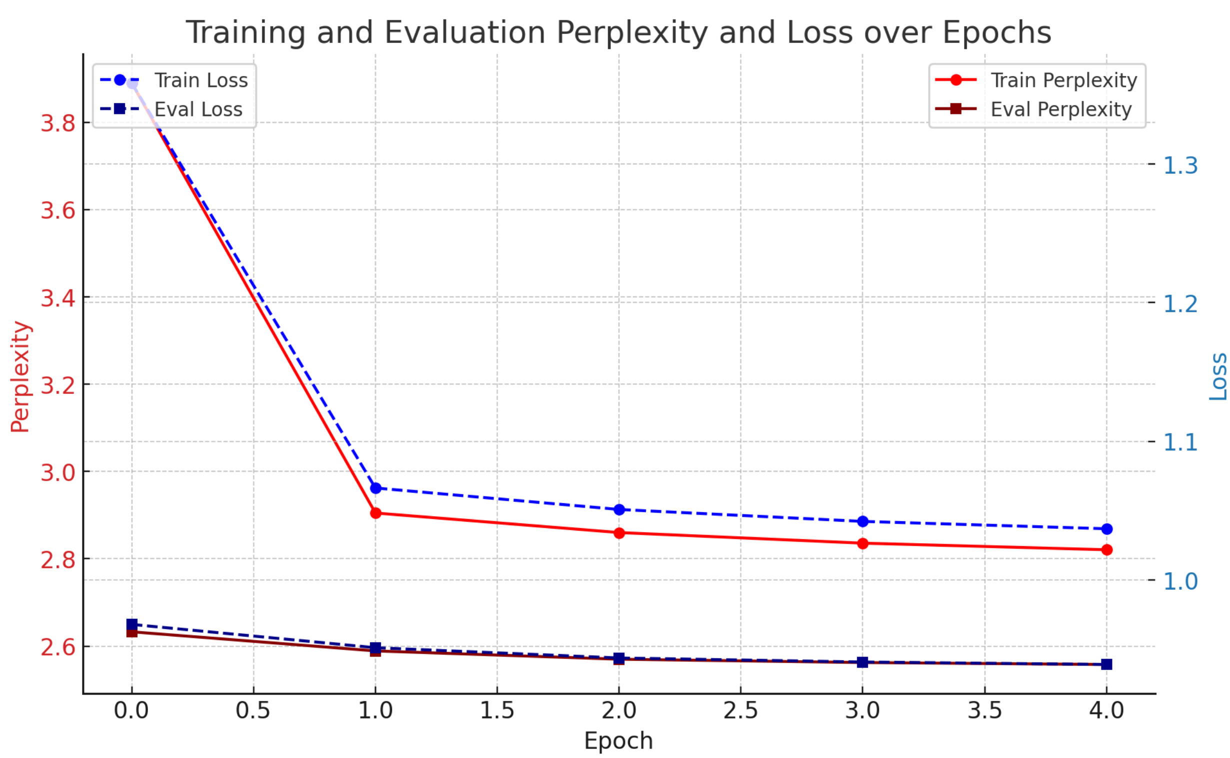


Fig. Training data of the T5 model with Prefix tuning.

The model achieved this with 5 epochs of training, lasting about 1 hour. Only a small fraction of the data available from the dataset was used. This demonstrated the efficacy of the fine tuning method when applied to the T5 model. As shown above, the training and evaluation loss drop quickly during the first epoch and then stabilize. **Perplexity** was also measured as it computes the uncertainty in predicting the next word in a given sequence. In this case we see that it gets lower as the training progresses, indicating that the model is improving.

```
{'log_norm': 4.840226220037563,
 'alpha': 4.209009679576707,
 'alpha_weighted': 13.421672312587289,
 'log_alpha_norm': 13.694369078535692,
 'log_spectral_norm': 3.259092615529022,
 'stable_rank': 53.740730533818756}
```

If we consider the weight distribution of the model, we can see that its alpha value is high indicating that the model is heavily tailed. This characteristic is often associated with models that have strong generalization capabilities.

In conclusion the prefix-tuning method has proven to be very effective for a complex task such as summarisation. This method could prove itself very useful for companies looking to **summarize complicated company meeting minutes**. It could be used to tailor the summaries for their case scenario instead of relying on other providers. For example, a company could train a model and use it locally without the need to upload sensitive data to third parties servers.

Llama-Adapter.

The **Llama-adapter is a lightweight fine tuning technique that is specific to the Llama models**. Meta released several of them in 2023, ranging from 7 to 70 billion parameters. Part of the innovation is the introduction of the instruction following task adaptation (Alpaca training[12]).Also, we have the introduction of the multi modal ability and the gating mechanism for gradual learning. The fine tuned models have several abilities (such as image recognition) which are added to the original model in the form of adapters. These abilities are combined to form a multi-ability model.

At a high level explanation, the model is fine tuned by appending a **learnable adaptation prompt** to the input and employing a **zero initialisation attention mechanism**. The attention mechanism is set to zero in order to gradually learn the additional parameters.

Mathematically, the Llama adapter can be expressed with this sequences of formulas:

$$T \in \mathbb{R}^{M \times C}$$

[12] Stanford university trained its own version of Llama with a dataset of instruction lead prompt https://crfm.stanford.edu/2023/03/13/alpaca.html

T represents the input sequence matrix and it is the **original input**. **M** denotes the sequence length and **C** represents the embedding sequence.

$$P \in \mathbb{R}^{K \times C}$$

P represents the **adaptation prompt** which is concatenated to **T** for each layer **L** where we want to apply the transformation. **K** Represents the length of the adaptation prompt.

$$T' = [P; T] \in \mathbb{R}^{(K+M) \times C}$$

The result **T**' is the concatenation of the prompt and the adaptation length is represented by (K +M)

The input sequence matrix **T** (that has the same dimension of original input by the embedding sequence) is concatenated to P or the adaptation prompt (the dimension is determined by the adaptation prompt multiplied by C the embedding sequence). This creates a new, longer sequence that combines both the prompt and the original sequence. The length of the new sequence is the total of the lengths of the prompt and the original sequence, however the width (embedding dimension) remains the same.

Another element of the Llama adapter is the attention mechanism set to zero for the layers where the adaption prompts are used. This is to minimize the impact of these prompts on the whole model. The zero setting is only applied on the modified layers and not the whole model. This is a way to introduce new information gradually into the model without overwhelming it. Most of the original model knowledge is preserved into the system with this targeted modification. **Training only the adapters and not the whole model is key to faster and cheaper fine tuning.**

The main characteristics of the Zero-initialized Attention with Gating mechanism are:

- **Zero Initialization**: To prevent disturbances, the attention is set to zero at the beginning of the training, it can be referred to as controlled introduction of new information. This gradual introduction might prevent gradient explosion that would compromise the calculation.
- **Gating**: A mechanism that controls the flow of information and it controls the importance of the new adaptation prompts during training. Therefore new information is added gradually as the training progresses. The gating mechanism penalizes underfitting prompts and gives more weights to better ones.

$$S_g^l = \left[\mathrm{softmax}(S_K^l) \cdot g_l; \mathrm{softmax}(S_{M+1}^l)\right]^T$$

The result of the formula represents the gated attention scores at a given layer of computation. On the left side, we have the normalized adaptation prompts scored through a softmax function and then multiplied by the gating factor (a learnable parameter). This result is concatenated with the normalized attention scores of the model's existing tokens. In a nutshell, this mechanism gives more importance to better-performing prompts, and it penalize underfitting ones, leading to gradual learning.

- **Separate Softmax**: original tokens and adaptation prompts are scored separately. This ensures that the two streams of data do not interfere with each other.
- **Stable Training and Progressive Learning**: The processes above ensure stability in the training while knowledge is updated progressively and not all at once.
- **Combination of pretraining and new signals**: Starting with the attention set to zero enables the model to combine its pre-existing knowledge with the instructions from the adaptation prompts. The results demonstrate a significant ability to adapt to new instructions.

Quoting from the original paper: *"With our proposed zero-initialized attention, the adaptation prompts can progressively inject the newly acquired instructional signals into the transformer, while simultaneously incorporating the pre-trained knowledge of LLaMA to provide high-quality responses*." Therefore the Llama adapter is a powerful combination of learnable prompts inserted between the frozen layers of the original model with a gating mechanism that controls the rate at which new information is progressively added.

Multiple Adapters.

One interesting characteristic of adapter modules is that several can be combined within the same model. For example, the researchers demonstrated that it is possible to combine visual and text input in one model. Their demonstration uses a visual encoder to extract information from images and integrates it with textual data for a question answering task.

$$I_p = \mathrm{Projection}\left(\mathrm{Concat}\left(\{I_m\}_{m=1}^{M}\right)\right)$$

The formula represents the projection of a feature vector derived from the concatenation of multi-scale features, transforming the multidimensional image vector into a single unified projection with the same dimensionality as the adaptation prompt.

$$P_l^v = P_l + \mathrm{Repeat}(I_p) \in \mathbb{R}^{K \times C}$$

The formula represents the multi modal incorporating the visual information. The vector Ip is repeated K times, so that its length matches the adaptation prompts, as they are required to maintain the same dimension.

The paper quotes the following *"In this way, LLaMA is fine-tuned to generate responses conditioned on vision-language inputs, and can tackle more challenging generative tasks with multi-modal understanding*."

Stanford Alpaca Training.

The researchers also included in the paper a practical training test where they used 52K instruction-following pairs for training in a json format. The training data is composed of sets of questions followed by the correct answer, designed to guide the model in following a specific structure. The researchers used OpenAI's text-davinci-003 model to produce the dataset[13]. An example of the dataset is represented below:

```
{
        "instruction": "Give three tips for staying healthy.",
        "input": "",
        "output": "1.Eat a balanced diet and make sure to include plenty
of fruits and vegetables. \n2. Exercise regularly to keep your body active
and strong. \n3. Get enough sleep and maintain a consistent sleep
schedule."
    },
    {
        "instruction": "What are the three primary colors?",
        "input": "",
        "output": "The three primary colors are red, blue, and yellow."
    },
```

Some of the specifications used in the training were: Llama-Adapter fine-tuned on 8 A100 GPUs for 5 epochs, and they used the smaller Llama7B model. The training cost less than $100.This demonstrates the efficacy of fine tuning a large language model of considerable size, both quickly and efficiently.
The researcher also used GPT 4 to evaluate the output of the Llama-Adapter fine-tuned model using 80 different questions. As indicated in their paper the model achieved more wins compared to Alpaca and Alpaca-LoRA, highlighting its effectiveness with zero-initialized attention mechanisms.

Colab Experiment of the Llama Adapter.

[13] https://crfm.stanford.edu/2023/03/13/alpaca.html

The code in the repository at https://github.com/OpenGVLab/LLaMA-Adapter was used to create a partial experiment on Colab.
https://colab.research.google.com/drive/1Q8S5OIC3PNmayZWCI1voLxLKZYliVEYF?usp=sharing
The code was originally designed to run on a Linux workstation, and requires a powerful GPU. The colab version required several adaptations in the terms of command execution that might prevent the code from running correctly. Also instructions are minimal and the code lacks clear comments. This has proved to be time consuming and challenging. However it was possible to launch the training loop and test it.
One additional interesting point is that the training loops uses **torchrun.** This is a utility provided by **PyTorch** for launching distributed training. Therefore, it leverages multiple nodes to perform computations in parallel, speeding up the process. However running this special routine on colab has to be investigated further.
The training loss is indicated as “**closs”**, it is not clear what it stands for. Whether it is indicating custom loss or something else the author should add additional information.

Fig. The training loss and learning rate extracted from the computation.

The **closs** experiences significant fluctuations. In any case, lit seems to be decreasing over time and stabilizing. This behavior could be attributed to the single epoch of training. The learning rate shows that it is progressively increasing, possibly a form of "warm-up" as the training loop indicates this feature to be active.

It was not possible to test this model during inference as it was not clear how to load the model back. Whether the weights needed to be extracted and used with the Hugging Face library alongside the original Llama model is unclear, as the exact procedure is not indicated anywhere.

In conclusion, it seems like a very promising method. However, Meta in fear of misuse has gated the models. It means that it requires registration. At the moment it seems impossible to register and obtain the model weights adding further difficulties. Despite the code being minimal, it lacks clear instructions. It is also lacking a tested colab notebook with examples. Practitioners might face challenges in implementing the solution on a desktop as they are required to have the same hardware. Colab often is the best alternative for those who do not have the resources for high end hardware.

Intrinsic Dimensionality (Aghajanyan et al., 2020).

Like the other method analyzed so far, this approach aims to reduce the number of trainable parameters in the model, while maintaining similar output. The author Aghajanyan et al. rightly discovered that these large language models might have a large amount of parameters but they models exhibit low intrinsic dimension in downstream tasks. This means that the parameters significantly contributing to the data are far fewer than expected. By targeting only these parameters we can reduce the computational requirement considerably.

What is Intrinsic dimensionality?

We can think of the **parameter space** of the model as a geometric object, we can say that the **intrinsic dimension** refers to the fact that these models might have a high dimension space, while, the significant parameters that capture the variability in the data are residing mainly on the surface of the high dimension object, with the interior largely empty. In other words, we can think of the intrinsic dimension as the **minimal number of parameters** (dimensions) required to achieve a performance comparable to training the entire model. Finally, we can say the model's effective parameters can be represented in a lower dimension space and still obtain the same performance.

Reparameterization Approach.

This method involved the reparametrization of the original model, from a higher space to a lower space dimension. We can express this concept with the following formula.

$$\theta_D = [\theta_0, \theta_1, ..., \theta_m]$$

Where θ_d expresses the parameter vector of the model, and D is the dimension.

$$\theta_D = \theta_{D_0} + P(\theta_d)$$

Here we have the original model in high dimension that is mapped with the function P to a lower dimension. θ_0 is the original parameters that is added to the function.

$$\theta_D = \theta_{D_0} + \theta_d M, \text{ where } M = HG\pi HB$$

Finally we can see the mapping function M, the paper specifies that the fastfood transform is used. The transform is made of several components.

H, a Hadamard matrix,
G, a diagonal matrix with Gaussian entries,
π, a permutation matrix,
B, a diagonal matrix with entries of ±1 with equal probability.

The fastfood transform approximates the kernel method. I**t helps to transform high dimension relationships in the data to a more linear space**. It is a **computationally efficient** method that could be used to approximate the minimum intrinsic dimensionality of a matrix. The combination of H,G,π, B allows us to generate a lower dimension space where we have **d<D**. The method does not directly discover the correct intrinsic dimension, but it helps with the reduction of the dimensions and therefore we obtain a matrix that is easier to compute. The fastfood transform is a tool for experimentation and the goal is to find a reduction of 90% of parameters, while maintaining performance comparable to the full model. In other words finding the correct **d** is an iterative process, we will continue reducing the dimensionality until we are satisfied with the output.

Unfortunately, it has proven difficult to implement an example of this technique with practical code. The only available implementation is at this URL https://github.com/rabeehk/compacter however, because of code obsolescence and outdated library dependencies, it does not run any longer. Upon inspection of **Intrinsic-SAID** (StructureAware Intrinsic Dimension) files, the configuration parameters indicate that the code used "fastfood" projection for dimensionality reduction. Other parameters indicated that the user could set the number of intrinsic dimensions as a tuning setting.

In the paper, the author recognizes that each part of the transformer model contributes uniquely to the overall output. When computing the intrinsic dimension for a model, each layer is treated separately, namely a **structure-aware approach.**

$$\theta D_i = \theta D_{0,i} + \lambda_i P(\theta_{d-m,i})$$

The structure-aware intrinsic dimension (SAID) for layer i corresponds to the calculation of the intrinsic dimension θD_0 (without structure adjustment) summed to the scaled structural parameter function (($P(\theta_{d-m,i})$ the dimensionality is reduced by m). P is a function that applied to the layers reduces its dimensionality. The formula allows the system to focus on layers that carry more relevant information.

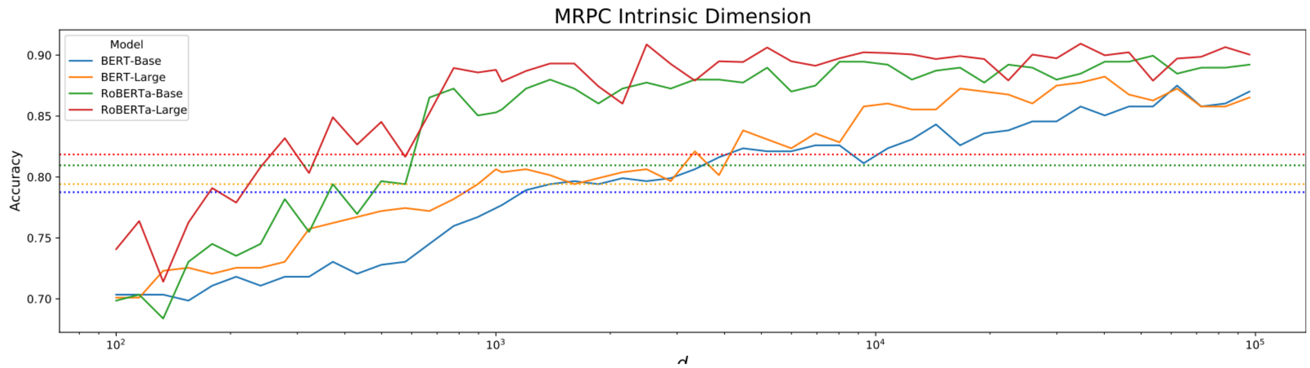


Fig. The amount of parameters needed to achieve a certain level of accuracy. Plot from the research paper.

Borrowing the plot from the research paper, we can see that as the researchers increase the number of dimensions, the accuracy also increases. For RoBERTa, the accuracy plateaus at around 10,000 parameters, indicating that **only a small number of parameters are needed to effectively represent the data**.

Quoting from the paper "The first takeaway is the incredible low dimensionality of viable solutions. With RoBERTa-Large, we can reach 90% of the full fine-tuning solution of MRPC using roughly 200 parameters and 800 parameters for QQP". Here the researcher has tested their solution using 2 datasets. For MRPC (Microsoft Research Paraphrase Corpus) where the fine tuned model output is compared to the paraphrased version. Their testing shows that they achieved 90% accuracy with only a handful of parameters. Considering that RoBERTa-Large has 355 millions parameters, this achievement is remarkable.

Intrinsic Dimension Experiment.

Plase see the colab at this URL
https://colab.research.google.com/drive/1avZo8fQ43dLxt33ugam08MdFUJd6tW7e?usp=sharing

While this experiment is not as nearly as sophisticated, it can help to better understand the implication of intrinsic dimensionality. Here a potential approach to explore is estimating the intrinsic dimensions of a model using **PCA**. Principal component analysis becomes inefficient when dealing with very high dimensionality. Therefore, combining the Fastfood transform to approximate kernel values with PCA, it helps to visualize the components explaining the most of the variance. With this in mind we can use other techniques like **node pruning** to reduce the complexity. The extent of pruning can be **empirically approximated** by examining the component analysis.

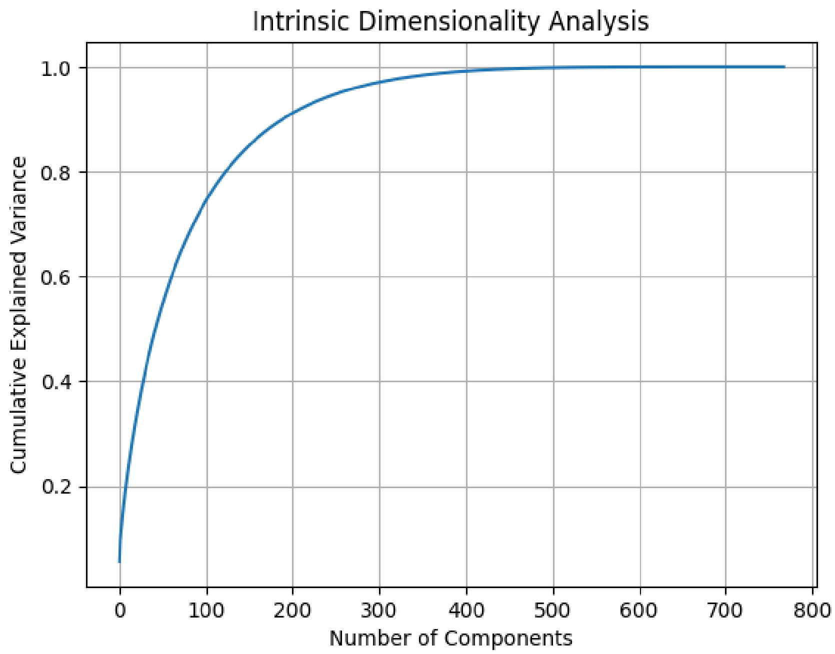


Fig. Fastfood and PCA applied to Bert embeddings.

In this short experiment, the **BERT** embeddings were loaded into memory, and then the **Fastfood** transform was applied. This method extracts linear relationships within the complex, high-dimensional data present in the embeddings. Then we can apply the **PCA** on the result of the Fastfood transform. The plot indicates that about 200 components explain most of the variance in the embeddings.This indicates that it is feasible to prune the embeddings without losing performance, thereby enhancing efficiency and resource utilization.

It is important to mention that pruning is a technique that sets a certain amount of weights to zero, technically excluding them from the computation. While this method does not inherently reduce the number of parameters, it allows skipping of certain connections in predefined layers of our choice. Furthermore pruning does not change the structure of the model but simplifies it. By empirical approach, we can test how much it is possible to prune without losing accuracy. Given that intrinsic dimension is the **minimum amount of parameters needed to explain the data**, if we could find a satisfactory pruned model, it is possible to argue that this technique also helps to explain what layers are most important in the model, and consequently find ways to optimize the neural network architecture.
In the colab experiment **xlm-roberta-base** was pruned of 2 millions parameters (focussing on the attention layers), not only the pruned model seemed to maintain the same output and accuracy but the inference time was considerably lower. Finally, a test for accuracy was conducted by loading the **wikitext** dataset. The pruned model received sentences from the dataset with masked words. Then the prediction of the mask is compared to the actual answer and accuracy measured. The pruned model showed an accuracy metric totally in line with the untouched model. Further confirming that the model is overparameterized and only a fraction of parameters actually contribute to the output.

The Tools For The Job: Investigation The Hugging Face Transformers PEFT Library.

With the introduction of the Hugging Face Transformers library, this repository and community have become the go-to resource for working with large language models. First of all, it serves as a repository for thousands of pre-trained models, and they can be imported with simple and easy commands. It also offers a unified **API** across **TensorFlow** and **Pytorch**, this means that developers can import the models in their favorite framework effortlessly. This is important because it improves accessibility of pre-trained models, and developers can use it without the need to relearn new tools or switch platforms.
More importantly, the Hugging Face transformers library already includes many methods for fine tuning of pre-trained models. They can be found under the PEFT framework. PEFT stands for parameter-efficient methods The PEFT Library is accessible from this url https://huggingface.co/docs/peft/index and it is rather extensive, covering several popular methods.

Some More Details About Hugging Face PEFT.

From the **transformers library,** we can easily import the language models. An example is provided in the **appendix** where GPT-2 is used, along with the **LoRa** method. In the **appendix**, we can also see the creation of the **LoRa configuration**.
It is important to declare the **task type**. For example, **CAUSAL_LM** will generate the next word in sequence such as creating a story from a seed sentence.

For more information on hyperparameters such as matrix rank, LoRA alpha, target modules, and dropout settings, as well as a **deeper dive into task types** and their impact on model fine-tuning, please refer to **Appendix** at the end of the document.

Chapter 3. The Task of Creating a Model Capable of Document Manipulation.

The main aim of this research is to develop a model using parameter-efficient techniques PEFT that runs on cloud-based hardware. The goal is to demonstrate that deploying intelligent applications in this manner can be cost-effective. The project seeks to show that individuals and small companies can benefit from fine-tuning open-source large language models. The demonstration also aims to streamline the process using cloud solutions that are cheap and available.

Details Extraction From Resume And Conversion in JSON format.

The problem: The avalanche of resumes[14] that the HR department started to receive in 2024 is overwhelming. This is a combination of several other changes that affect the recruiting process in the post pandemic era. In 2024, job seekers have been sending more applications because of the perceived uncertainty caused by the cost of living. As people feel insecure, they are trying to get a better position. Secondly, the increased use of streamlined online processes (LinkedIn's 'Easy Apply' feature) leads to a massive amount of applications being sent.

Now it is typical for a company, when recruiting, to deal with a large number of unstructured documents. It has been the duty of the HR department to sift through hundreds of resumes, one by one. It is a very time consuming job. This is a massive hurdle for companies to stay afloat in the game by selecting the right candidates while filtering out the unrelated applications. Previous attempts involved asking candidates to fill in forms. This would give some structure to the data, but it also deters candidates as 92% of job seeker do not complete the application process[15].

Large language models can help us with this task and sift through this amount of unstructured data, and create some structure automatically. A pre-trained LLM could be finetuned using the LoRa technique to “digest” documents automatically and output the needed information into a JSON format, streamlining the process and saving valuable time for HR professionals.
JSON file representation is similar to Python dictionaries. In a nutshell it is a data structure with

---

[14] https://www.visual-planning.com/en/blog/hr-statistics-and-trends
[15] https://www.visual-planning.com/en/blog/hr-statistics-and-trends

nested parent and child elements, allowing for hierarchical organization of data. This format is widely used by many applications enabling efficient data manipulation and retrieval.

The process would involve **identifying a suitable pretrained model,** creating the necessary **dataset** of resumes along with their JSON extraction. Then the model would need to be finetuned, tested, and deployed for use.

Finally, the model will be deployed within containerized applications, accessible through RESTful API endpoints. This approach ensures that these models can be leveraged as intelligent systems able to understand and process text effectively. Once the application is containerised, it can be easily accessed and used for inference via standard internet protocols, enabling interaction from any system capable of making **HTTP requests**.

The steps above would be a potential pipeline for the deployment of fine-tuned models in cloud applications. The advantage of this demonstration is that we are training the model with boundaries within a certain knowledge. The model will use its language understanding capabilities, but the output will be limited by the JSON format. To expand this further, the model is being asked to identify skills, or experience within a text and capture them in a particular format.
Why is this choice relevant for this project? This task is simpler than creating a chatbot that answers questions, as it focuses on structured data extraction rather than dynamic interaction based on vast knowledge. This approach can significantly streamline HR processes and while it demonstrates the capabilities of the aforementioned approach.

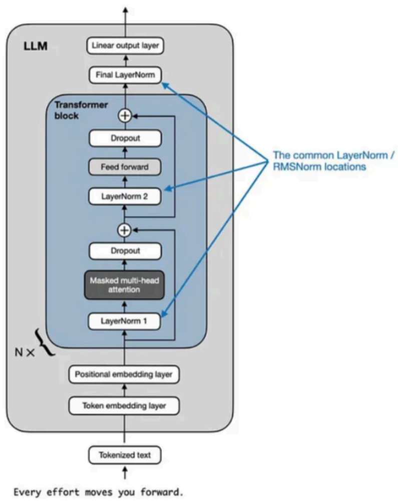
Methods of Resume To JSON Extraction.

In the following chapter, a detailed pipeline for the creation of this text manipulation application will be presented. This includes:

- **Model Choice**: Given the rapidly evolving ecosystem of machine learning models, it is crucial to invest time in identifying and describing the optimal choice of models for a specific task.
- **Data acquisition and manipulation** in order to create a dataset that is suitable for our scope. A dataset of resumès is needed along with the Json representation.
- **Model trainin**g. A suitable pretrained model will be fine-tuned with one of the techniques explained earlier. The aim is to train it cheaply and quickly.
- **Deployment of the model in a cloud application**. A user can input a resume and receive back its JSON representation,

By following this simple pipeline it is possible to produce fully functional models to extract information from resumes, automating extraction of candidate information.

Google Gemma As Base Model.



	Gemma			
Parameters	2B	7B	Llama 2 7B	OLMo 7B
d_model	2048	3072	4096	4096
Layers	18	28	32	32
Feedforward hidden dims	32768	49152	11008	8192
Num heads	8	16	32	32
Num KV heads	1	16	16	16
Head size	256	256	128	128
Vocab size	256128	256128	32000	50280
Exact parameter number		9,324,112,896	6,738,415,616	6,888,628,224
Accounting for weight tying		8,537,680,896		

multiquery attention

Extremely large vocabulary size

$$\mathrm{GEGLU}(x, W, V, b, c) = \overbrace{\mathrm{GELU}(xW + b)}^{\text{Sigmoidal part}} \otimes \overbrace{(xV + c)}^{\text{Linear projection}}$$

$$\mathrm{SwiGLU}(x, W, V, b, c, \beta) = \mathrm{Swish}_{\beta}(xW + b) \otimes (xV + c)$$

Fig. The Gemma architecture[16]

Google has introduced **Gemma**, a new family of lightweight, state-of-the-art open models that are free to use. In reality, the wording “open” does not mean that it is **open source** but rather **open-weight**, meaning the model's weights or pre-trained parameters are available, but not the source code or training data.

The choice is centered around the Google Gemma models. These LLMs are generally a compromise between both the older and smaller GPT models and the larger and more sophisticated Llama 2 models. **Gemma has 2 billion parameters,** and the designer claims it is able to run it on portable devices such as phones and laptops. Furthermore, Gemma models perform very well on most benchmark testing even when compared to much larger models. For example, even the smaller **Gemma 2b** performs very well in the **HellaSwag** test, where the model must choose plausible continuation from a set of options.

[16] Gemma Architecture https://medium.com/@shravankoninti/gemma-introducing-new-state-of-the-art-open-model-by-google-caae9fe29972

		LLaMA-2		Mistral	Gemma	
Benchmark	metric	7B	13B	7B	2B	7B
MMLU	5-shot, top-1	45.3	54.8	62.5	42.3	**64.3**
HellaSwag	0-shot	77.2	80.7	81.0	71.4	**81.2**
PIQA	0-shot	78.8	80.5	**82.2**	77.3	81.2
SIQA	0-shot	48.3	50.3	47.0*	49.7	**51.8**
Boolq	0-shot	77.4	81.7	**83.2***	69.4	**83.2**
Winogrande	partial scoring	69.2	72.8	**74.2**	65.4	72.3
CQA	7-shot	57.8	67.3	66.3*	65.3	**71.3**
OBQA		**58.6**	57.0	52.2	47.8	52.8
ARC-e		75.2	77.3	80.5	73.2	**81.5**
ARC-c		45.9	49.4	**54.9**	42.1	53.2
TriviaQA	5-shot	72.1	**79.6**	62.5	53.2	63.4
NQ	5-shot	25.7	**31.2**	23.2	12.5	23.0
HumanEval	pass@1	12.8	18.3	26.2	22.0	**32.3**
MBPP†	3-shot	20.8	30.6	40.2*	29.2	**44.4**
GSM8K	maj@1	14.6	28.7	35.4*	17.7	**46.4**
MATH	4-shot	2.5	3.9	12.7	11.8	**24.3**
AGIEval		29.3	39.1	41.2*	24.2	**41.7**
BBH		32.6	39.4	**56.1***	35.2	55.1
Average		47.0	52.2	54.0	44.9	**56.4**

Fig. The Performance of Gemma Models.

Besides the fewer parameters, the Gemma architecture offers several improvements. It is a heavily optimized model with several enhancements. For example, the **multi-query attention** mechanism reduces computational complexity and improves speed by enabling more efficient parallel processing.
In the case of multi-query attention, K and V are the same for all heads and it only computes a new Q for each head[17]. In the model diagram we can see that "Num heads" is 8, and the "Num KV heads" is 1, **indicating that there are 8 query heads but only 1 shared key-value set for all those heads**. There is a trade-off in terms of the potential for a slight decrease in model performance and accuracy, but the simplification of the attention mechanism leads to much faster computation. Therefore, this trait may be beneficial in scenarios where computation constraints are a concern.

RoPE embeddings[18]are shared across inputs and outputs, which enhances model efficiency and reduces memory usage. Also, with this method instead of using traditional sinusoidal functions, the embeddings received their positional encoding via **rotational matrix transformation**. With this method, it is possible to encode the position of the embeddings using a matrix, which has proven to increase the computational efficiency[19].

[17] Multiquey attention https://paperswithcode.com/method/multi-query-attention
[18] RoPe Embeddings https://paperswithcode.com/method/rope
[19]RoFormer: Enhanced Transformer with Rotary Position Embedding
https://ar5iv.labs.arxiv.org/html/2104.09864

GeGLU Activations are more efficient than the classic **ReLU** activation[20].

$$\mathrm{GeGLU}(x, W, V, b, c) = \mathrm{GELU}(xW + b) \otimes (xV + c)$$

The GeGLU activation is a complex function that offers a combination of two other functions: the Gaussian Error Linear Unit (GELU) and the Gated Linear Unit (GLU). The activation function, in a nutshell, **uses a gating mechanism to decide how much information should flow through**.

GeGLU(x, W, V, b, c), where x is the input, W and V are learned matrices, computed during the **gating** and activation process. The parameter b and c are two bias vectors learned at the training stage. The first part, GELU(xW + b), represents the Gaussian Error Linear Unit applied to x with weight matrix W and bias b.
This function introduces **stochastic elements into the computation**, it makes the activations follow a probability distribution. **The probabilistic non-linearity output helps the model to learn complex patterns more effectively**. Finally, the last part, (xV + c), represents an element-wise multiplication and xV + c is a linear transformation of x (where V is a weight matrix and c is a bias vector). It is a linear projection enhancing the model's capacity to process and adapt to incoming data.

Additionally, **RMSNorm, or Root Mean Square Normalization,** is more efficient than the normalization method used in the original transformer architecture, leading to faster and more stable training. The commonly used **layer normalization** in neural networks aims to scale the output to achieve consistent variances and means across different instances, which speeds up the learning process. **Root Mean Square Normalization is more efficient** as it eliminates the need to compute and subtract the mean. In fact, this function achieves normalization by calculating the root mean square value, further simplifying the process.

With this brief introduction to Gemma, we can reaffirm that it is a strong candidate for fine-tuning due to its outstanding capabilities and relative computational efficiency. The model is expected to train quickly and cost-effectively with our custom data.

[20]geglu https://paperswithcode.com/method/geglu

Chapter 4 Implementation Details.

The Dataset Preparation.

The link to the data preparation and model inference in colab can be found here:

https://colab.research.google.com/drive/1I1xXHDK-32fSjYPSvScJ6cRJ7cfa-QOs?usp=sharing

We must not forget that when fine-tuning a generic large language model, the dataset for our domain specialization plays a pivotal role in determining the final outcome. Often it is more important than all the hyperparameters combined. We should expose our model to enough examples in order to obtain meaningful results. For example, if our downstream task is to fine-tune a model to extract data from resumes and convert it to JSON format, we should prepare a dataset that conditions the model for this particular output.

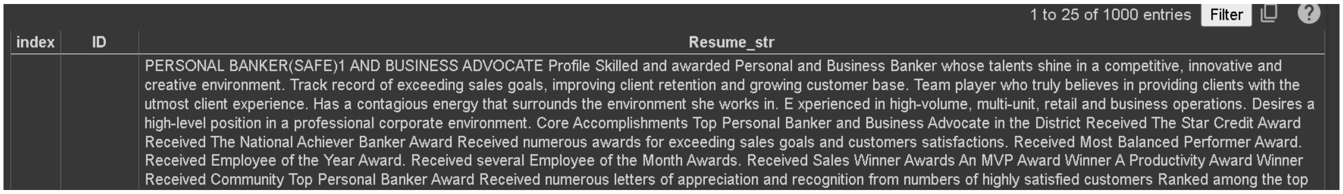
First, we should dive into the task of material gathering; for example, we could collect a variety of resumes from different industries and formats. However since this is extremely time consuming and there is the convenience of a ready made dataset, we should take advantage of this. The **Resume Dataset from Kaggle** can be exploited for this task**.** The dataset is labelled as: **A collection of Resumes in PDF as well as String format for data extraction**[21], making it an ideal resource for training our model.
The dataset, curated by **Snehaan Bhawal**, contains a collection of resumes from various industries. It is specifically designed for tasks such as text extraction and natural language processing. The dataset includes multiple resume samples, which can be used to train models for parsing and extracting structured information like personal details, education, work experience, and skills into JSON format. **This dataset is ideal for developing and fine-tuning models for automated resume screening and analysis.**

Since the original dataset contains almost 2500 entries, a subset was created with 1000 entries. This number of examples should be enough for the fine tuning task. Additional columns were

---

[21] Resume dataset https://www.kaggle.com/datasets/snehaanbhawal/resume-dataset

also removed, finally the dataset now looks like this:



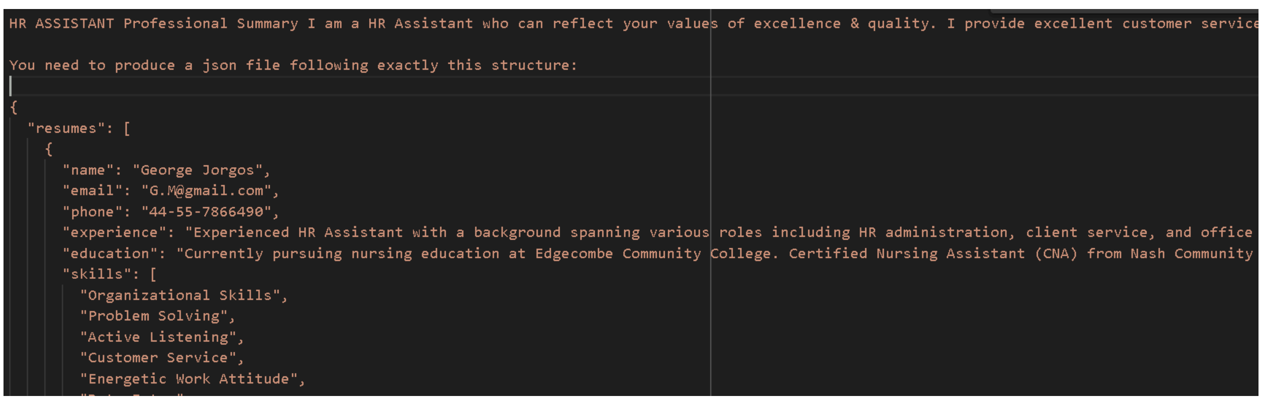
1 to 25 of 1000 entries Filter

index	ID	Resume_str
		PERSONAL BANKER(SAFE)1 AND BUSINESS ADVOCATE Profile Skilled and awarded Personal and Business Banker whose talents shine in a competitive, innovative and creative environment. Track record of exceeding sales goals, improving client retention and growing customer base. Team player who truly believes in providing clients with the utmost client experience. Has a contagious energy that surrounds the environment she works in. E xperienced in high-volume, multi-unit, retail and business operations. Desires a high-level position in a professional corporate environment. Core Accomplishments Top Personal Banker and Business Advocate in the District Received The Star Credit Award Received The National Achiever Banker Award Received numerous awards for exceeding sales goals and customers satisfactions. Received Most Balanced Performer Award. Received Employee of the Year Award. Received several Employee of the Month Awards. Received Sales Winner Awards An MVP Award Winner A Productivity Award Winner Received Community Top Personal Banker Award Received numerous letters of appreciation and recognition from numbers of highly satisfied customers Ranked among the top

Fig. Upon opening of the dataset in Pandas dataframe we can see the resumè

Each entry contains an anonymous resums with a lengthy description, however this representation is not sufficient for the task. The correct representation should be as follow:

- **Each entry must contain the resume in its entirety**
- **Each entry is followed by the resume extraction in JSON format**
- **The JSON format it well formed and correct**
- **The JSON representation contains the correct extraction from the resume**



```
HR ASSISTANT Professional Summary I am a HR Assistant who can reflect your values of excellence & quality. I provide excellent customer service

You need to produce a json file following exactly this structure:

{
  "resumes": [
    {
      "name": "George Jorgos",
      "email": "G.M@gmail.com",
      "phone": "44-55-7866490",
      "experience": "Experienced HR Assistant with a background spanning various roles including HR administration, client service, and office
      "education": "Currently pursuing nursing education at Edgecombe Community College. Certified Nursing Assistant (CNA) from Nash Community
      "skills": [
        "Organizational Skills",
        "Problem Solving",
        "Active Listening",
        "Customer Service",
        "Energetic Work Attitude",
```

Fig. A Complete training example.

Our goal here is to create a number of correct examples, and use them for fine tuning Gemma. Therefore we need a curated set of examples that carefully represent the desired output. Theoretically, this task should be done by a team of human labellers that carefully read each resume and create the corresponding extraction.

Since this is an expensive and lengthy task, another approach was taken:

- **A single example was created with the correct output.**
- **GPT3 Turbo was used to generate the rest of the representations.**
- **The output was added to the data frame ready for further processing**

GPT3.5 by OpenAI is intelligent enough to accomplish this task quickly and cheaply and the result is excellent. In reality this output should be still verified by a human annotator to make sure that extractions are truly correct.

	ID	Resume_str	Generated_JSON
0	40088790	PERSONAL BANKER(SAFE)1 AND BUSINESS A...	{\n "resumes": [\n {\n "name": "Jane ...
1	16861758	HR ASSISTANT Professional S...	{\n "resumes": [\n {\n "name": "HR As...
2	28973180	SALES / FINANCE MANAGER Summary...	{\n "resumes": [\n {\n "name": "John ...
3	17033567	VIDEO DIRECTOR, EAST COAST VIDEO FOR ...	{\n "resumes": [\n {\n "name": "John ...
4	49119887	MARKETING & PUBLIC RELATIONS MANAGER ...	{\n "resumes": [\n {\n "name": "John ...

Fig. The automatically annotated data frame with the Json data extraction.

The final step is to create examples of single strings that will be used as training examples. This is achieved by concatenating the various rows of the dataframe.

```
df['text'] = '###resume: ' + df['Resume_str'] + ' ###json: ' + df['Generated_JSON']
```

Fig. the concatenation of chunks of text with the addition of the prefix

To guide the model in transforming resumes into JSON format, specific markers are added to indicate where different sections start and end. For instance, by prefixing an unseen resume with ###resume and ending it with ###json, the model can recognize the boundaries of the input text. Given that the models are autoregressive, this will prompt them to generate the JSON output following the resume text, ensuring the correct output format.

Model Training.

Fine-tuning the model using the **PEFT** library from Hugging Face is the most crucial step of the project. The process involves configuring numerous parameters for both model and hardware configuration. To speed the process up, it is possible to use a web interface of a containerised training application from **Hugging Face**. The application is called **Auto Train** and it abstracts most of the intricacies. In a nutshell, it allows the user to select supporting hardware such as a powerful GPU. This hardware is hired by the minute and turned off automatically at the end of the training, preventing unnecessary charges.

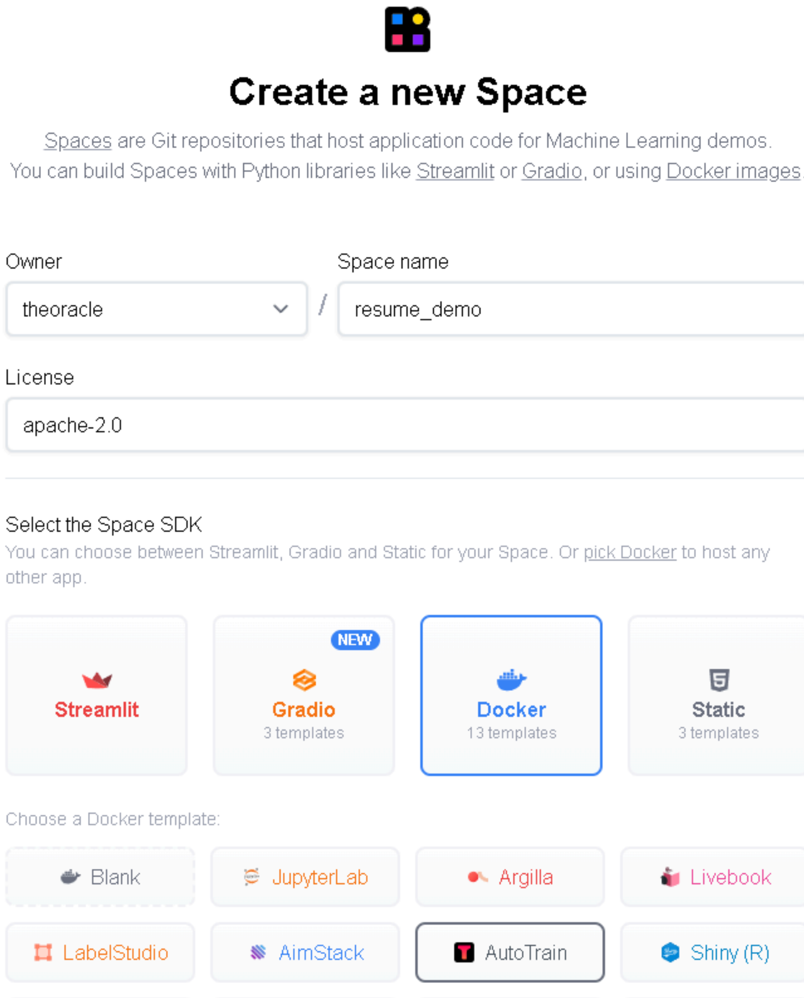


Fig. The creation of the Docker auto train application from HF

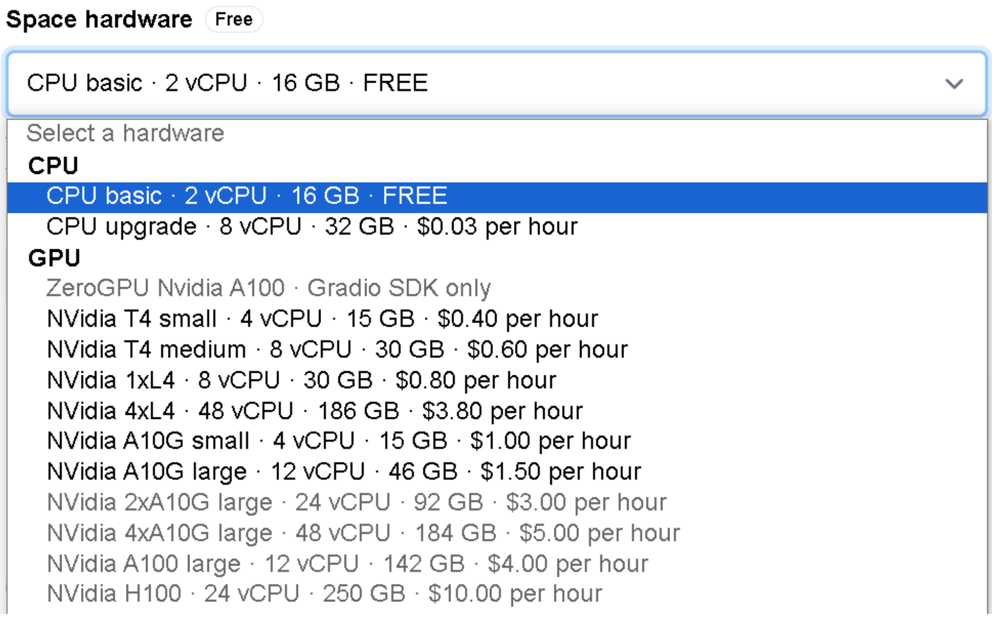


Fig. The selection of hardware that we can choose and use by the minute.

As shown above, there is a wide selection of GPUs available for the training job. In this case, the NVIDIA T4 is sufficient for the required computations. This choice allows us to leverage parallel processing, significantly accelerating the calculations.

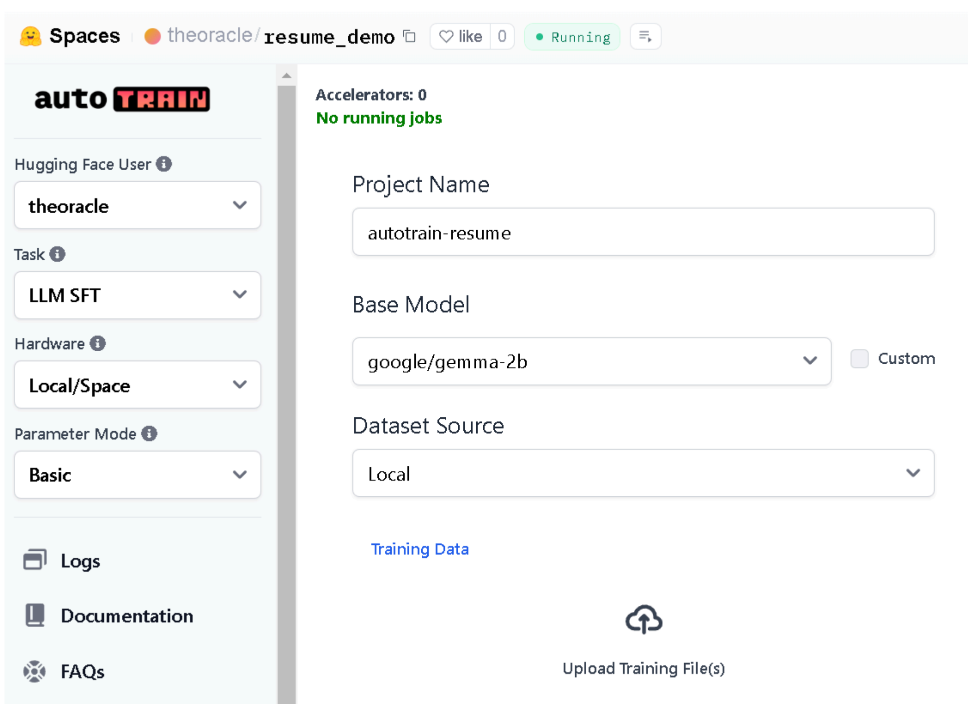


Fig. The interface of the PEFT Fine Tuning application

Once the containerised application is created, the interface is automatically launched. Here, it is possible to set the basic configuration such as project name, model choice, and most importantly upload the csv file that contains our training data.

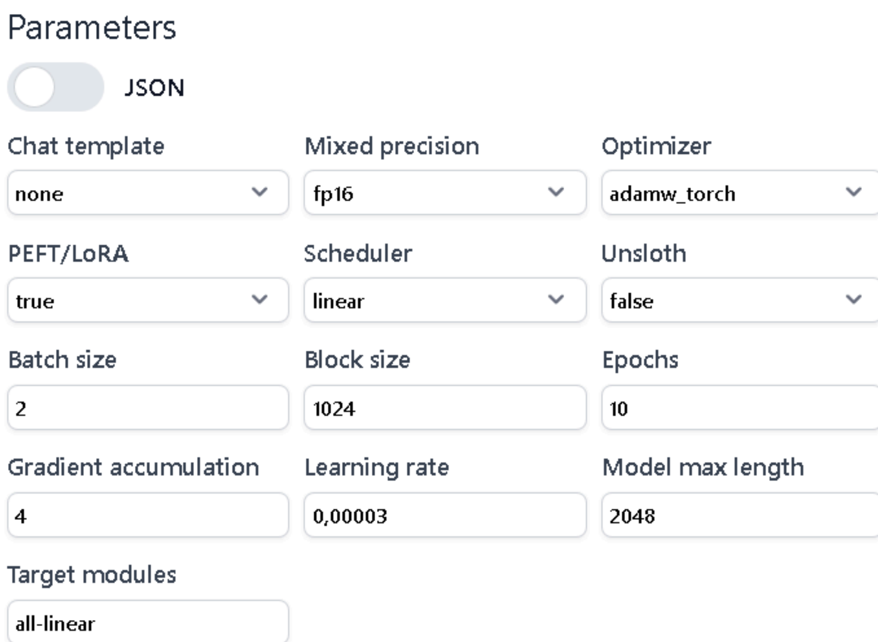


Fig. These are the training parameters of the model that we tune if needed.

In the same interface, we can set the various hyperparameters. The most important one is the number of epochs, as this determines how many times the model will see the entire dataset. **In this case 10 was the choice**, this means that the model will be trained 10 times on the entirety of the data. The key is finding the optimal balance for training. Too few epochs, and the model won't learn effectively; too many, and it might overfit, memorizing the training data rather than generalizing.

Other elements that play a crucial role are:

Mixed precision (fp16): Uses half-precision (16-bit) floating-point for faster computation and lower memory usage. By decreasing the number of the floating points we decrease the precision of the computation, however in many cases it does not matter as the output can still be as good as using more bits and this loss of precision doesn't significantly impact the final output.

Optimizer: Algorithm used to adjust the model's parameters. Adam (Adaptive Moment Estimation) is an adaptive algorithm that can change the learning rate according to the gradients

of the loss function. Therefore we do not need to spend time to find the correct learning rate as it will be adjusted for us, reducing the need for manual tuning.
PEFT/LoRA: Parameter-efficient fine-tuning or **Low-Rank Adaptation**. The interface allows you to choose this method for efficient model fine-tuning.

Learning rate: Speed or the step at which the model learns. In this case we choose a low starting value and the optimizers will change dynamically according to the training.

Batch size: Number of samples processed before the model is updated. Choosing larger batches leads to out of memory errors, and smaller batchers lead to longer training time. **For this example, a batch size of 1 or 2 is sufficient.**

Gradient accumulation: Number of batches to accumulate before updating the model's parameters. To avoid constantly updating the gradient at each batch, we can choose to update less frequently. Accumulating more batches at once will increase memory usage, while accumulating fewer batches will lead to longer computation times.

Scheduler: Method to adjust the learning rate during training (e.g., linear). The scheduler modifies the learning rate at specific intervals. By choosing linear, it gradually adjusts the learning rate linearly over time.

Model max length: This parameter defines the maximum length of the input sequences that the model can process. Similarly to other parameters longer sequences are more computationally expensive and take longer time to process. This parameter is set based on the available hardware.

Chapter 5 Task Result and Analysis.

Hugging Face AutoTrain simplifies the training process by automatically logging training data. This is a crucial element as we can see how the training went and it allows us to review the training process and draw conclusions.

Gradient Normalization.

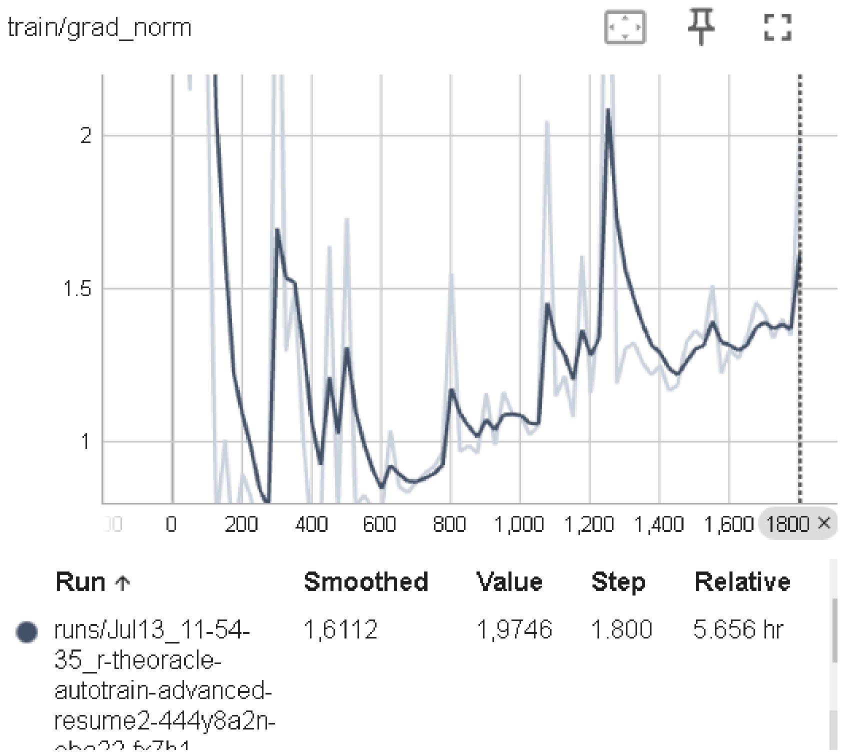


Fig. Gradient normalization from training.

Gradient normalization is a technique to adjust the gradients during the training of a neural network to improve stability and performance. It rescales the gradient to have a consistent norm. It is crucial because it prevents issues such as exploding or vanishing gradients.
The plot from the model training shows a high gradient normalization that drops quickly, this is typical, and it signifies that the model started to converge. However it increases again over time, with several fluctuations. This is ground for investigations, in future optimisations a different starting learning rate can be considered.

Learning Rate.

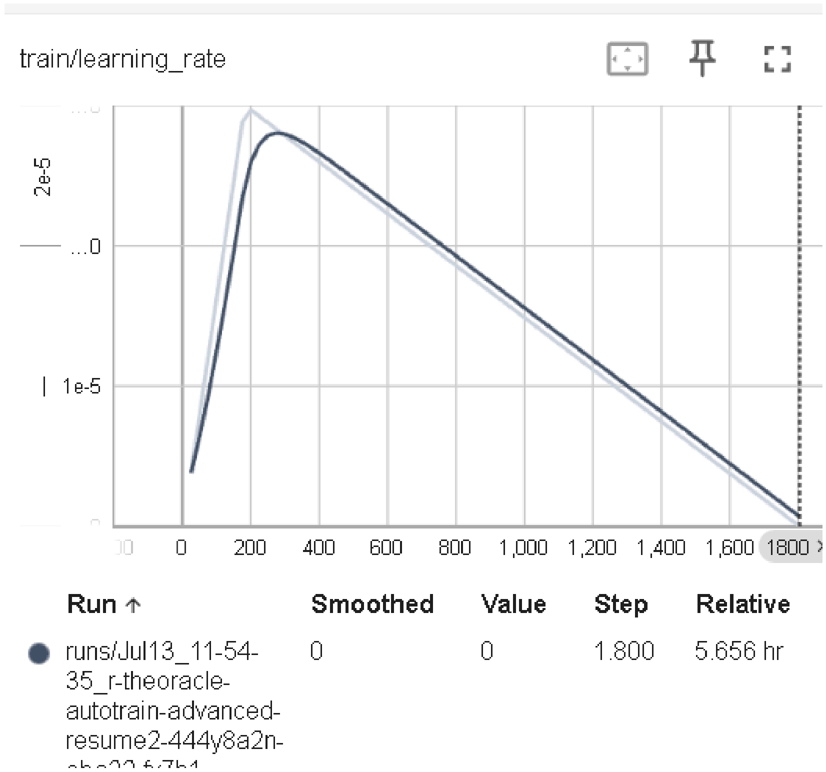


Fig. The learning Rate from the training

The learning rate is a step or an adjustment applied to the gradient update. It is crucial because if it is set too high the training might overshoot the minimum of the error function and lead to poor convergence. In this case, because we are using an adaptive learning rate through ADAM, it seems to work smoothly. Initially it increases abruptly however this is typical as the algorithm makes larger updates at the beginning then it slowly starts to converge.

Loss.

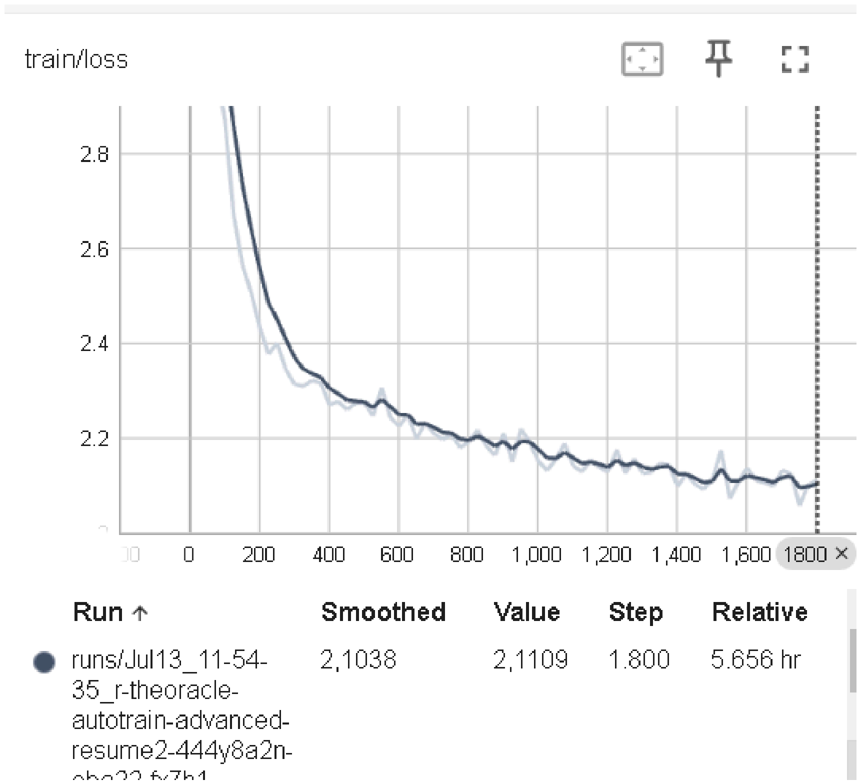


Fig. Training loss.

Training Loss: During this process it's important to assess what we have actually achieved. The training is normally over a series of epochs where the model is trying to learn the representation of the dataset. After each epoch, the model's performance is evaluated by calculating the loss. This represents **the error or in other words how distant is our prediction from the desired result.** The values do not represent an empirical measurement, but rather they are a representation of how well the model is learning.

Typically, datasets are split in train and evaluation. This is a standard approach in machine learning where we reserve a part of our dataset for a later evaluation. The split can be for example 90% for the train split and 10% for the evaluation split.

However in this case the entire dataset was used and no split for evaluation was available. This is because the training dataset was limited to 1000 examples, and the experimentative nature of the project required usaging of all examples to rapidly obtain results at a low cost.
We can see that the loss initially drops and then smoothly decreases over time, indicating that the model is learning and converging. However without validation, it is difficult to detect if the model is overfitting. Overfitted models tend to generalize poorly

Manual Evaluation.

The model is publicly available at the following url https://huggingface.co/theoracle/resume2

Finally the fine tuned model using **LoRa** has completed its training and it is now possible to manually evaluate it with a few examples. To sum up, the model should take a resume that it has not seen before and extract the content in formatted JSON. In the attached Colab several experiments were conducted and all seem to have output the correct JSON that is both well formed and contains all the relevant information from the resume.

```
prompt4='''
**Lena Rivera**
Email: lena.rivera@email.com
Phone: 987-654-3210

I am currently serving as the Chief Technology Officer at Innovatech Solutions, a position I've held since January 2018, based in San Francisco,

I founded GreenCode, a non-profit organization dedicated to teaching coding skills to underprivileged youth, in March 2015. Through this initiat

My academic background includes a Master’s degree in Computer Science from MIT, completed in 2013, where my research focused on artificial intel

My technical skills are deeply rooted in programming languages such as Python, Java, and C++. I am proficient in managing both SQL and NoSQL dat

Languages: I am fluent in English and Spanish, with full professional proficiency in both.
###json:
```

Fig. A prompt with the resume information for the model.

When the fine tuned model is prompted with the resume it outputs the following:

```
{"resumes": [
    {
      "name": "Lena Rivera",
      "email": "lena.rivera@email.com",
      "phone": "987-654-3210",
      "experience": "Experienced Chief Technology Officer with a
background in software development and cloud computing. Skilled in
project management, team leadership, and the development of
innovative software solutions. Previous roles include Senior Software
Engineer at TechAdvance and Founder of GreenCode.",
      "education": "Master of Science in Computer Science from MIT in
2013. Bachelor of Science in Software Engineering from University of
California, Berkeley in 2010.",
      "skills": [
        "Python",
        "Java",
        "C++",
        "SQL",
```

```
        "NoSQL",
        "AWS",
        "Project Management",
        "Strategic Planning",
        "Team Leadership",
        "Coding Skills",
        "Real-Time Systems",
        "Artificial Intelligence",
        "Machine Learning",
        "Cloud Computing"
      ]
    }
  ]
}
```

The model shows that it is able to understand both natural language and the formatting. Not only is it able to understand how to associate the contact details with the correct labels but also it is able to complete all the other fields. For example the education and the experience fields are filled correctly. Furthermore the model has summarized the text and not just copied it over. The fields include all the important details in a summarized version. **Furthermore it understands the meaning of “skills”, filling the right field correctly.**

To conclude, the experiment has been mostly successful in conditioning a pre-trained model to extract data from casually formatted resumes. Certainly this model is not production ready. A company interested in deploying such a model in production should conduct large scale automated testing and human evaluation.

However this demonstrates that a recruiting agency, looking to automate the CV screening process, can automate this task with Gemma. The pretrained model is conditioned with an example dataset of resumes and JSON examples. The training happens over the **Hugging Face Auto train which simplifies the process**. Especially by abstracting the complicated hardware setups. With an Auto Train we can utilize the Transformer library and the LoRa technique simply and efficiently. LoRa is able to reduce the complexity of the attention matrices, speeding up the model training considerably. The fine-tuning described above only took a couple of hours and the cost totalled to only $1.50. This is a hands-on experiment, demonstrating that small entities can leverage the capabilities of LLMs at very reduced cost.

Weight Watchers Library For Further Model Evaluation.

Another interesting evaluation technique to adopt is from a library called **WeightWatcher**[22]. This library checks the **eigenvalue distribution** of the model's weights. Nicely and correctly trained models have a **heavy tail distribution.** This is indicative of values clustered around the mean, but the overall weight distribution extends into a long and heavy tail. This suggests a diverse range of values, meaning the model has seen a good variety of data and is therefore able to generalize well.

```
{'log_norm': 1.6533578949584984, 'alpha': 5.305512742383686,
'alpha_weighted': 2.9621337380084163, 'log_alpha_norm':
3.164429125774414, 'log_spectral_norm': 0.4268605097633952,
'stable_rank': 59.99080779789094}
```

For example: the above shows the analysis of the parameters of the weights of the LoRa trained model for the resume task. These are **empirical values** that can give an idea of the model's ability to generalize well and not **overfitting**.
In this case the model is somewhat between the ability of generalizing well and somewhat not prone to overfitting. To have an idea, we can say that if we see **a very low log norm we have a model with low weight values, indicating underfitting**. On the other hand with **higher positive values** we can say that the model has **large weight values therefore it is more able to generalize well** with unseen data. In a nutshell **a log norm of 1.653358** is not excessively high, indicating that the model's weights are within a reasonable range.

For an in-depth explanation of metrics such as Alpha, Log Alpha norm, Log Spectral norm, and Stable Rank, as well as insights on under-trained and over-trained layers, please see the Appendix at the end of the document.

Chapter 6 Containerisation Of The Model For Inference.

As we have worked hard to create a model, it is now time to deploy it. The model can be containerised with "**all it needs to run**" (all necessary components such as libraries and dependencies) in an "**image**",or often referred to as a container via the Docker Engine.
Docker containerisation is a way to **pack the model with all the dependencies and libraries** in a single lightweight and **immutable container** that will boot fast and it will always work. Containers will not suffer from "**but it worked on my laptop**" syndrome because it is packing all the correct libraries in a single immutable environment. In other words, the container will

[22] The weight watcher library https://pypi.org/project/weightwatcher/

encapsulate the application, its dependencies, and runtime environment, and it will always work even in different computing environments.
As a result, our application is now **very portable and easy to deploy.**
Additionally our container also includes the FastAPI Python library that can help us to create API endpoints. In the context of **APIs (Application Programming Interfaces**) the endpoints in our application are specific paths or **URLS that can handle requests** such as the resume to JSON extractions.

Hugging Face Spaces.

For convenience the model's **Docker image** will be hosted on **Hugging Face Spaces**. It offers a simple solution to attach several kinds of GPU to the computing space. It is a more straightforward solution than those proposed by Google Cloud or Amazon. Beside the complicated interface and cumbersome setting, Google and Amazon cloud seem to **suffer from a chronic lack of GPUs** which lead to time consuming research for the correct hardware[23]. On the other hand, Hugging Face offers solutions entirely dedicated to machine learning with total abstraction of hardware intricacies, with high availability of hardware, therefore it is the place to go at the moment.
After creating the space itself, it is possible to start working on the configuration files immediately. **Let's see how the Docker image is composed:**

```
$ ls
.gitattributes
Dockerfile
README.md
main.py
requirements.txt
static/
```

DockerFile. The Dockerfile is one of the most important elements. It is a script that contains a series of commands to create an image for a Docker container. It defines the environment in which an application runs and the necessary dependencies.

main.py. This is the main Python script of the application. It contains the logic and functions that load the model and control the API endpoints.

Requirements.txt. This file lists all the dependencies and libraries needed to run the Python application. Here we will load the **Transformers** and the **PEFT** libraries.

---

[23] GPU Poor
https://fixyourfin.medium.com/the-impact-of-gpu-shortages-and-cloud-costs-on-ai-industry-e96f29833c7f

static/. This is a directory that contains 2 files: **index.html and script.js.** These files are responsible for rendering the user interface on the web.

Python Script And API Endpoint In More Details.

```
from fastapi import FastAPI, Query
from starlette.responses import FileResponse
from fastapi.staticfiles import StaticFiles
from fastapi.responses import JSONResponse

from transformers import AutoModelForCausalLM, AutoTokenizer

@app.get("/gemmac64")

def gemma_64(input):#: str = Query(default="")
    #device = "cuda" if torch.cuda.is_available() else "cpu"

    prompt = f'''###resume: {input}
    ###json: '''

    # Tokenize the prompt to get input IDs and attention mask
    encoding = tokenizer(prompt, return_tensors='pt', padding=True,
truncation=True, max_length=500, add_special_tokens=True)
    input_ids = encoding['input_ids']
    attention_mask = encoding['attention_mask']

    # Generate output, ensuring to pass the attention mask and set
pad_token_id
    output_ids = model.generate(
        input_ids.to('cuda'),
        attention_mask=attention_mask.to('cuda'),
        max_new_tokens=300,  # Specify the number of new tokens to generate
        pad_token_id=tokenizer.eos_token_id,
        #temperature=0.7,
        #do_sample=True,
        #top_k=1,
        #repetition_penalty=1.5

       )
    response = tokenizer.decode(output_ids[0], skip_special_tokens=True)
```

```
    return JSONResponse(content={"output": response})
```

This is the part of the code that creates the pacific endpoint for model inference. In this case the **FastAPI** library is used. It is a modern, fast (high-performance), web framework for building APIs with Python 3.7+.

In our case, the GET endpoint **endpoint /gemma_64** takes an input query parameter from the user (from a web form in this case), such as the resume text. Then it tokenizes the input prompt. It generates a response using the model, it decodes the generated tokens into a readable string and returns it as a **JSON response**.

The Web Interface.

The containerized application can be accessed from the web through the **FastApi endpoint**, a simple HTML and Javascript for rendering the page and allowing the user to paste a resume text.
This is a simple demonstration of how the application works, allowing users to interactively convert their resumes into JSON format through an easy-to-use web interface. The containerized application ensures portability and scalability, making it easy to deploy on various platforms.

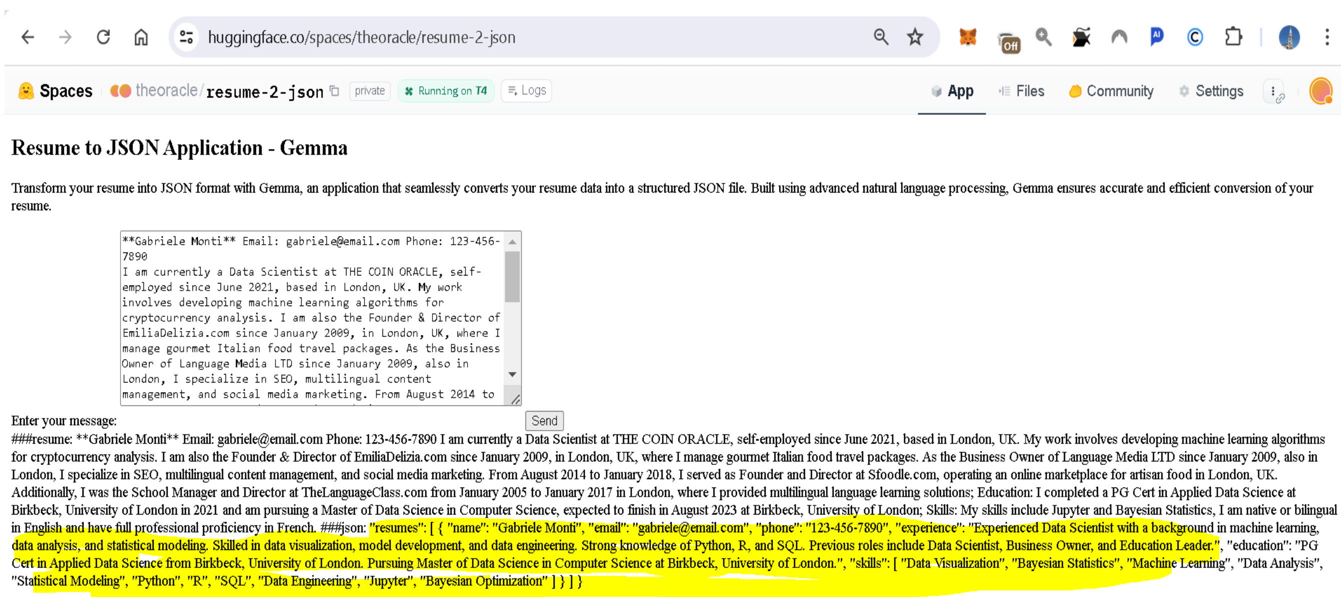


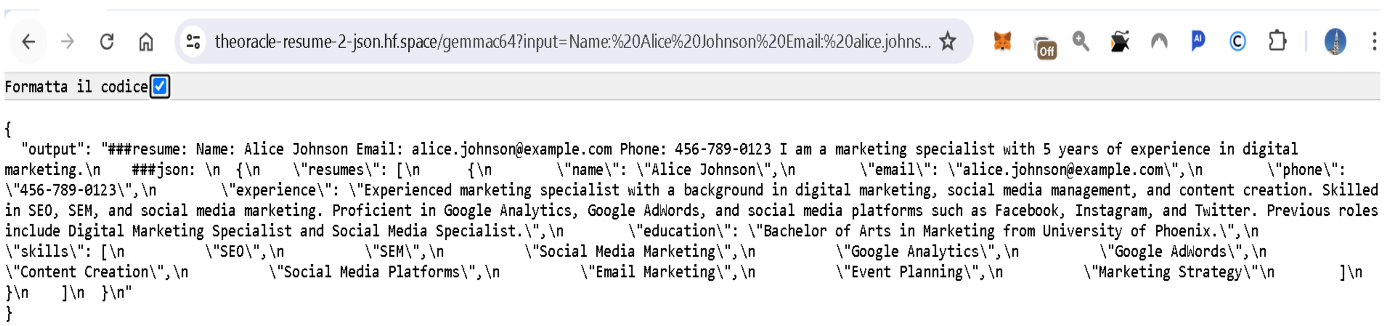
Fig. The web interface can process resmè text and return the Json format to the user.

RestFul API Address.

The containerized application has the following API address https://theoracle-resume-2-json.hf.space/gemmac64. This takes advantage of standard HTTP requests. This means that developers do not need to install additional software or libraries when using the RESTful API. We can send a standard GET request (get data from a specified resource) to this URL and attach resume data to the same URL. For example:

```
https://theoracle-resume-2-json.hf.space/gemmac64?input=Name: Alice Johnson
Email: alice.johnson@example.com Phone: 456-789-0123 I am a marketing
specialist with 5 years of experience in digital marketing.
```

When the request is received by the application, the system will return the resume's JSON representation generated by the model. This JSON output includes structured data extracted from the provided resume text


theoracle-resume-2-json.hf.space/gemmac64?input=Name:%20Alice%20Johnson%20Email:%20alice.johns...

Formatta il codice


```
{
  "output": "###resume: Name: Alice Johnson Email: alice.johnson@example.com Phone: 456-789-0123 I am a marketing specialist with 5 years of experience in digital marketing.\n    ###json: \n  {\n    \"resumes\": [\n      {\n        \"name\": \"Alice Johnson\",\n        \"email\": \"alice.johnson@example.com\",\n        \"phone\": \"456-789-0123\",\n        \"experience\": \"Experienced marketing specialist with a background in digital marketing, social media management, and content creation. Skilled in SEO, SEM, and social media marketing. Proficient in Google Analytics, Google AdWords, and social media platforms such as Facebook, Instagram, and Twitter. Previous roles include Digital Marketing Specialist and Social Media Specialist.\",\n        \"education\": \"Bachelor of Arts in Marketing from University of Phoenix.\",\n        \"skills\": [\n          \"SEO\",\n          \"SEM\",\n          \"Social Media Marketing\",\n          \"Google Analytics\",\n          \"Google AdWords\",\n          \"Content Creation\",\n          \"Social Media Platforms\",\n          \"Email Marketing\",\n          \"Event Planning\",\n          \"Marketing Strategy\"\n        ]\n      }\n    ]\n  }\n"
}
```


Fig. The output of the model after getting a GET request from the API URL.

RESTful Apis are particularly well suited for this application because the endpoint completely abstracts the workings behind the application including all the code. We can just supply the user with the API URL and they can attach their own data for processing. Therefore the resume processing API application can be the backbone for other applications. These programs can take large amounts of resumes and further process them.

This application and its **RESTful** API demonstrate how a containerized LLM fine-tuned application can automate large-scale data processing. For example, it simplifies the recruitment process in human resources by automating the conversion of resumes into structured JSON format, making it easier to manage and analyze large volumes of candidate information.

Chapter 6 Challenges Of The Project and Conclusion.

The challenges encountered in this project are many; however, three can be pinpointed as the main challenges going forward:

1. **Lack of properly formatted data.** This is probably the main challenge of all data science projects. Often we have all the tools for the jobs but we lack the data in suitable format. In this case it was important to have a properly annotated dataset for the **JSON** representation. This task should be done manually, or at least it should be a verifiable process. In production, all necessary steps should be taken to compile a foolproof set of examples that can be used for training the model. In this project, it was demonstrated that it is possible to leverage other language models to generate or manipulate data. However, a verification process should be in place to check the output.

2. **Model hallucinations**. The model trained in this experiment demonstrates the capabilities of the model in understanding human language and it is able to format text correctly. However, if it receives an incoherent input from the user (something totally unrelated), it will still try to generate the **JSON** file with output based on its training. Therefore, the system should have a process in place to stop the user from inserting data other than resume text. Otherwise the system should have a step where the user approves the output before insertion into a final database.

3. **Complexity of Working with LLM**: Large Language Models (LLMs) are complex to work with due to their vast size and the computational resources required for training and inference. This requires engineers with a wide set of skills, from data compliance and manipulation, to deployment of models at scale via containerized application. Also, projects require careful management by experienced staff, which might be out of reach for small entities. However, the use of the aforementioned fine tuning techniques has demonstrated that it is not necessary to invest large amounts of capital.

4. **Lack of properly working code examples.** While very promising, the **Llama adapter** and **intrinsic dimensionality** projects did not really have working examples available for the public. Often, these projects rely on other libraries and have a lot of dependencies. This means that as these libraries are updated, the code relying on them could stop working within days of release. The process of updating the code and fixing the dependencies could be very time consuming. It is worth considering spending this time exploring more documented approaches. Also, it must be noted that very often research papers do not come with code implementation, and the training data. Therefore, replicating the exact result is not an easy task.

Final Thoughts And Conclusions.

It is now clearer that lightweight fine tuning of open source large language models will help with the democratization of the technology, as a consequence we will see a wider adoption among private users and companies. This can help innovation across industries, providing solutions that were previously out of reach. It will lower entry barriers for startups that want to dive into this new technology, creating a vibrant ecosystem that can complement the market dominated

by large corporations.
This will also help to better understand what lies behind closed-source models like ChatGPT. Researchers using open-source models can gain deeper insights into this technology, leading to increased transparency and interpretability of the final results. By studying and improving open-source models, the research community can foster innovation, enhance model reliability, and ensure ethical use.
This goes without mentioning that lightweight finetuning is much more resource efficient than the whole model fine tuning. This translates to a reduced carbon footprint, faster deployment to production, and of course huge savings in terms of costs.

LoRa Vs Other Methods.

The **LoRa** method seems to have gained a special place in the fine tuning hall of fame. In the resume section LoRa was specifically selected as it appears as the main method in the hugging face PEFT library. Let's analyze why it is so popular.

LoRa seems to favor **retention of pre-trained knowledge[24],** this seems a valid reason to choose this method over others. **Catastrophic forgetting** happens when a model loses all knowledge during the fine-tuning process. This leads to the model to lose all its capacity and perform very poorly. **LoRa** seems to prevent this problem by freezing most of the model and compressing only the target layers.
Another reason why it is a favorable method is that it is often used in conjunction with **quantisation.** While it is difficult to pinpoint, both methods used together seem to introduce a synergic effect and deliver further computational efficiency.
For a deeper analysis of related methods such as QLoRa, AdaLoRA, LoHa, LoKr, and VeRa, please refer to the Appendix.

In conclusion, it is possible to argue that LoRa and, secondly, the adapter techniques are currently the most efficient and reproducible methods available today. While other methods, such as intrinsic dimensionality, are promising, the lack of code and data does not allow for consistent and reliable results

References:

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., ... & Rush, A.M. (2019). HuggingFace's Transformers: State-of-the-art Natural Language Processing. ArXiv, abs/1910.03771. This paper introduces the Hugging Face Transformers library.

[24] https://heidloff.net/article/efficient-fine-tuning-lora/

Vaswani, A. et al. (2017). Attention is all you need. In Advances in Neural Information Processing Systems. Introduces the Transformer model, fundamental for modern large language models.

Edward J. Hu et al. (2021). LoRA: Low-Rank Adaptation of Large Language Models.. This paper introduces the LoRA technique, focusing on the adaptation of large language models through low-rank matrices to achieve efficient fine-tuning.

Zhang, R., Han, J., Liu, C., Gao, P., Zhou, A., Hu, X., Yan, S., Lu, P., Li, H., & Qiao, Y. (2023). LLaMA-Adapter: Efficient Fine-tuning of Language Models with Zero-init Attention. arXiv preprint arXiv:2303.16199v2. https://doi.org/10.48550/arXiv.2303.16199

Houlsby, N. et al. (2019). Parameter-Efficient Transfer Learning for NLP. In Proceedings of the 36th International Conference on Machine Learning. Discusses adapters for efficient NLP model transfer learning.

Li, L., & Liang, P. (2021). Prefix-Tuning: Optimizing Continuous Prompts for Generation. arXiv:2101.00190. Presents Prefix-Tuning for fine-tuning language models with continuous prompts.

Aghajanyan, A., Gupta, S., & Zettlemoyer, L. (2021). Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning. arXiv:2105.14103. Investigates how few parameters can significantly impact model behavior.

The code in the colab
https://colab.research.google.com/drive/1avZo8fQ43dLxt33ugam08MdFUJd6tW7e?usp=sharing
Has been **generated with the help of ChatGpt** with the prompt:

Prompt: Implement a function for the Fast Walsh-Hadamard transform in Python using NumPy or Scipy. It should work on vectors and utilize Hadamard matrices.

Appendix.

This appendix contains additional information and examples on topics discussed earlier in this document.

Tokenization example

To demonstrate tokenisation we can start with the following example

"Hello, world! This is an example."

The vector of the tokenisation could be like this below, where [CLS] represents the beginning of the sentence and [SEP] is the beginning of the following sentence. This is particularly true in the BERT transformer (Bidirectional Encoder Representations from Transformers) where the beginning of each sentence is identified. Each architecture would have its own way to create the tokens.

["[CLS]", "Hello", ",", "world", "!", "[SEP]", "This", "is", "an", "example", "."]

The numeric representation is assigned to each part of the sentence:

{"[CLS]": 0, "Hello": 1, ",": 2, "world": 3, "!": 4, "[SEP]": 5, "This": 6, "is": 7, "an": 8, "example": 9, ".": 10}

The representation could be something like this

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Each token is mapped to a higher dimension vector such as

0 -> [0, 0]
1 -> [0.1, 0.1]
2 -> [0.2, 0.2]
3 -> [0.3, 0.3]
.
.

The final token embedding could be represented like this:

[[0, 0], [0.1, 0.1], [0.2, 0.2], [0.3, 0.3], [0.4, 0.4], [0.5, 0.5], [0.6, 0.6], [0.7, 0.7], [0.8, 0.8], [0.9, 0.9], [1.0, 1.0]]

This is a simplified example, in practice the actual vectors have much higher dimensions, in order to capture the complex relationships necessary for language processing tasks.

One Hot Encoding Example.

A simple example of one hot encoding of a typical example: CAT SAT ON THE MAT

```
The: [1, 0, 0, 0, 0,0]

cat: [0, 1, 0, 0, 0,0]

sat: [0, 0, 1, 0, 0,0]

on: [0, 0, 0, 1, 0,0]

the: [0, 0, 0, 0, 1,0]

mat: [0, 0, 0, 0, 0,1]
```

Then given this sentence we can create a context window C=2. This means that given a word we take the 2 next words around it. For example, we take THE, our C will be (THE, CAT) and (THE, SAT). Otherwise, if we take CAT we will have (THE, CAT), and (CAT, SAT) and (CAT, ON)

The neural network learns the weights by minimizing the difference between the prediction and the target word via a loss function such as cross entropy, then the output is predicted by using **softmax function** for the final prediction which outputs either a 1 or 0 accordingly.
With this method we can indeed produce a set of weights from the network that present the target word in relation to all other word combinations. This is the actual aim of the embedding to capture the relationship that exists in the context.

The Q,V,K Matrices And Their Mathematical Representation

This section provides an example matrix representation of Q,V,K. These matrices contain weights that are learned during the traditional computation of a neural network with gradient updates through backpropagation. The training depends on the problem that we want to solve.

In our case scenario, we might train a model to predict the next word in a sequence. The **input could be “this envelope has no…”** and the desired result could be **“seal”**. The weights are initialized with random values and updated iteratively. The update happens when the output is compared with the desired word **by computing the error** (or the difference between the desired result and the output). At this stage the weights are updated through the calculation of the gradient in respect to each weight. Therefore Q,V,K matrices are able to learn different representations of the sentence in relation to the desired output. These matrix representations are then used for our linear transformation when we compute the attention.

$$W_Q = \begin{bmatrix} w_1 & w_2 & w_3 & \cdots \\ w_4 & w_5 & w_6 & \cdots \\ w_7 & w_8 & w_9 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$W_K = \begin{bmatrix} w_{10} & w_{11} & w_{12} & \cdots \\ w_{13} & w_{14} & w_{15} & \cdots \\ w_{16} & w_{17} & w_{18} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$W_V = \begin{bmatrix} w_{19} & w_{20} & w_{21} & \cdots \\ w_{22} & w_{23} & w_{24} & \cdots \\ w_{25} & w_{26} & w_{27} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Q, V, K are initialized with different random weights and once fully trained each has a different purpose. As we discussed earlier, **Q represents the question, K represents the "information” or what we are trying to retrieve. Finally V represents the actual content that will be retrieved**.

Here below we can see a matrix dot multiplication where x_1, x_2…are the embeddings and w_1, w_2… are the learned weights of the Q,K,V matrices.

$$Q = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \cdots \\ x_{21} & x_{22} & x_{23} & \cdots \\ x_{31} & x_{32} & x_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} w_1 & w_2 & w_3 & \cdots \\ w_4 & w_5 & w_6 & \cdots \\ w_7 & w_8 & w_9 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$K = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \cdots \\ x_{21} & x_{22} & x_{23} & \cdots \\ x_{31} & x_{32} & x_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} w_{10} & w_{11} & w_{12} & \cdots \\ w_{13} & w_{14} & w_{15} & \cdots \\ w_{16} & w_{17} & w_{18} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$V = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \cdots \\ x_{21} & x_{22} & x_{23} & \cdots \\ x_{31} & x_{32} & x_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} w_{19} & w_{20} & w_{21} & \cdots \\ w_{22} & w_{23} & w_{24} & \cdots \\ w_{25} & w_{26} & w_{27} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Some More Details About Hugging Face PEFT.

This appendix provides a detailed explanation of key concepts related to the Low-Rank Adaptation (LoRA) method, including important hyperparameters such as matrix rank, LoRA alpha, target modules, and dropout rates

One of the main hyperparameters is the **rank of the A and B matrices** that will be multiplied together to obtain the original matrix. This metric specifies the resolution of the data and it will influence the amount of trainable parameters of the model. Finding the right rank is a work of trial and error, as the aim is to reduce the dimension while keeping an acceptable output.
Lora Alpha is the scaling factor. This indicated the magnitude of the low rank updates when applied to the original matrix. Therefore this influences the **strength of these updates**, consequently we need to carefully tune this factor.
The **Target Modules** are the actual part of the model that we want to target, in the code examples we are targeting the following: **["attn.c_proj", "mlp.c_proj"]**. In other words it targets the projection layers in the attention mechanism (c_proj) and the MLP layers (c_proj) or also called the Multi-Layer Perceptron. In this example, these layers will receive the modifications from the application of the LoRa matrices.
The lora dropout is another hyperparameter that we can use at our advantage. It is there to prevent the model from overfitting. In the example 0.1 means that 10% of the the outputs in the LoRA-adapted layers will be zeroed out randomly. This regularization technique will prevent the model from relying too much on specific adaptation.

In conclusion we can test the **PEFT** techniques using this handy library, even though it hides most of the complexity of the job, it could be very useful to quickly test these methods. The library dramatically cuts prototyping time, by providing a layer of abstraction through simple to read code.

```python
from transformers import GPT2Model, GPT2Config

from peft import LoraConfig, get_peft_model

# Step 1: Define the configuration for LoRA
lora_config = LoraConfig(
    task_type="CAUSAL_LM",  # Specify the task type, e.g., causal language
modeling
    r=8,  # Rank of the low-rank matrices
    lora_alpha=32,  # Scaling factor for LoRA
    target_modules=["attn.c_proj", "mlp.c_proj"],  # Target specific
modules in GPT-2
    lora_dropout=0.1,  # Dropout rate for LoRA layers
)

# Step 2: Load the base GPT-2 model
model_name = "gpt2"  # You can specify other model sizes like "gpt2-medium"
base_model = GPT2Model.from_pretrained(model_name)
```

Additional Information About The Task Type.

It is also worth spending some time to understand the **task type function** that is added to the parameters, as it matters when fine-tuning the pretrained models. For example in **CAUSAL_LM** the model has access only to the tokens before the word is trying to predict. Therefore the processing is unidirectional (left to right) and sequential. While when using **SEQ_2_SEQ_LM** is used for translation the processing is substantially different. In this case the model has access to **the entirety of the text that** it is trying to translate.
SEQ_CLS is used for classification tasks, where the model given a sentence outputs only a label.
QUESTION_ANS is a task that extracts the answer from a given text, and could be particularly useful to extract correct information without mistakes.
Considering the above, the LLM practitioner should be aware of the task is trying to accomplish, as this might not always be clear at the beginning. **Tasks might be ambiguous** for example a chat bot might seem to be causal generation but answering questions might be a better choice. There is a great deal of overlapping abilities in the pre-trained models, lack of clear instructions or opaque pre-training (we have no access to the original training data). Therefore it is often difficult to determine what a model is actually capable of. Choosing the right task type is a challenge.

Additional Information about the Weight Watcher Test On the Trained Model.

In this appendix, we go deeper into several key metrics that are essential for evaluating model performance and understanding training dynamics.
A **low alpha suggests a fatter heavy-tailed distribution**, in the context of a neural network is something that can indicate that the **network has capacity for complexity**, and it is expected to perform well with unseen data. **Alpha metric keeps into account the importance of the magnitude of the eigenvalues of the matrix**, this can also be an indicator for overfitting/underfitting factors of the large language model. An alpha value greater than 4.0 often indicates undertraining, suggesting that the model might not have learned sufficient patterns from the data. **Here, an alpha of 5.305513 implies potential undertraining.**

Alpha_weighted value. The value close to 3.0 suggests a moderate level of learning. Here, it shows that while the model may be under trained in some aspects, it still has captured some meaningful patterns.

The **Log Alpha norm** takes the log of the Alpha distribution. This also helps to understand the variability of the data but since we might be dealing with many different scales, the log helps to provide a more normalized measure. Indicating the diversity and spread of the eigenvalues, in this case log_alpha_norm of 3.16 suggests a wide spread. **This means that the model has learned a variety of patterns but also could indicate instability in learning.**

The **log spectral norm** can help us to understand the ability of amplification of the network. Low amplification is preferable as the ability to amplify signal and noise is generally limited, this avoids problems like exploding gradients or amplification of noise. **Here, 0.426861 is relatively low, indicating good stability.**

Another interesting metric is the **stable rank**. A high stable rank means that the matrix is hardly compressible and it does not have so much redundant data. Therefore is less prone to overfitting.This might be an interesting metric as we are looking to compress redundant data in the large language models. **A value of 59.99 is moderately high, indicating that the model is effectively utilizing its capacity.**

Finally the presence of **155 under-trained layers and 113 over-trained layers** in the model indicates an imbalance in the layers. Further refining and investigation could be necessary to create a better model. Since only 1000 examples were used in the training, augmenting the training dataset could be a good idea to immediately produce a better model.

Additional LoRa Methods

In this appendix we cover additional LoRa techniques that highlight the strength of the method. For example the **QLoRa**[25]method takes advantage of both methods under one roof, leading to faster deployment of fine-tuned models.
Finally these methods can only survive and prosper if there is a large community supporting them. Which not only **LoRa** seems to have but also it has given rise to several other spin offs. For example **AdaLoRA**[26] with its adaptive rank selection. As LoRa focuses in the same way to all layers, basically using the same rank for all of the targets, AdaLoRA can adaptively adjust the rank during training. This means that it can use resources for layers with higher rank and use less for less lower rank matrices. While maintaining all the benefits of LoRa, this newer method potentially uses even less resources and faster training.
LoHa[27] is another method that uses the **Hadamard** product to approximate the LoRa matrices. This method approximates the large matrix by finding the LoRa adaptation and combining them back into one by element wise multiplication (Hadamard product). This seems to reduce the number of parameters further while maintaining the same accurate representation.
Similarly **LoKr**[28] approximate the 2 low ranking matrices into one using the Kronecker product. This calculation involves taking each element from the first matrix and multiplying it by the entire second matrix. This produces an efficient representation of the initial matrices.
While **LoHa** and **LoKr**[29] are applicable to large language models at the moment the only implementation available seems to cover stable diffusion.
Finally **VeRa**[30] Random Matrix Adaptation which has an implementation on Hugging Face for LLMs**.** This method reduces the number of parameters **dramatically** (with consistent output) by using the same low rank matrices for all layers and computing only 2 two additional vectors per layer. The method implementation seems to have some limitations to linear layers but it could be a promising method where hardware and hardware constraints are an issue.

[25] QLoRa https://training.continuumlabs.ai/training/the-fine-tuning-process/parameter-efficient-fine-tuning/qlora-and-fine-tuning-of-quantized-language-models-lms
[26]AdaLoRA https://huggingface.co/docs/peft/package_reference/adalora
[27] LoHa https://huggingface.co/docs/peft/package_reference/loha
[28] LoKr https://huggingface.co/docs/peft/package_reference/lokr
[29] LoRa variants for LLM https://gautam75.medium.com/exploring-different-lora-variants-for-efficient-llm-fine-tuning-4ca41179e658
[30] VeRa https://huggingface.co/docs/peft/package_reference/vera

Additional Resume Tests.

In this section I have created resumes using chatgpt 4o and I have used the fine tuned model to extract the data in order to have further testing examples.

Prompt: **Generate a concise resume for Arda Yılmaz, an experienced Goldfish Breeder with over 12 years of expertise, detailing skills, background, and qualifications in breeding and maintaining goldfish.**

Arda Yılmaz | 123 Fish Lane, Aquatown, FL 33001 | ardayilmaz@example.com | (123) 456-7890 | **Experienced Goldfish Breeder** with over 12 years of experience in breeding, raising, and maintaining a variety of goldfish species. Proven expertise in creating optimal aquatic environments to ensure the health and vitality of fish. Committed to providing high-quality goldfish to hobbyists and retailers. **Background:** Aquatown Goldfish Farm, Aquatown, FL - *Goldfish Breeder* (January 2012 - Present): Successfully bred multiple species of goldfish, maintained breeding logs, ensured optimal tank conditions, and managed feeding schedules. Consistently achieved high survival rates and vibrant coloration in goldfish. Provided customer education on goldfish care. **Qualifications:** High School Diploma from Aquatown High School. Certified in Aquaculture Management. **Abilities:** Breeding Techniques, Water Quality Management, Fish Health Monitoring, Customer Service, Tank Maintenance, Aquatic Plant Care, Problem-solving, Detail-oriented.

The fine tuned model returned this Json output:


```
  "resumes": [
    {
      "name": "Arda Yılmaz",
      "email": "ardayilmaz@example.com",
      "phone": "123-456-7890",
      "experience": "Experienced Goldfish Breeder with over 12 years of
experience in breeding, raising, and maintaining a variety of goldfish
species. Proven expertise in creating optimal aquatic environments to
ensure the health and vitality of fish. Committed to providing high-quality
goldfish to hobbyists and retailers.",
      "education": "High School Diploma from Aquatown High School.
Certified in Aquaculture Management.",
      "skills": [
```

```
        "Breeding Techniques",
        "Water Quality Management",
        "Fish Health Monitoring",
        "Customer Service",
        "Tank Maintenance",
        "Aquatic Plant Care",
        "Problem-solving",
        "Detail-oriented"
      ]
    }
  ]
}
```


It has extracted the data correctly however sometimes it seems to miss the opening curly parenthesis at the beginning of the text.

Now I ask chat GPT to compare the original text and the extraction in a table.

Field	Original Text	JSON
Name	Arda Yılmaz	Arda Yılmaz
Email	ardayilmaz@example.com	ardayilmaz@example.com
Phone	(123) 456-7890	123-456-7890
Experience	Experienced Goldfish Breeder with over 12 years of experience in breeding, raising, and maintaining a variety of goldfish species. Proven expertise in creating optimal aquatic environments to ensure the health and vitality of fish. Committed to providing high-quality goldfish to hobbyists and retailers.	Experienced Goldfish Breeder with over 12 years of experience in breeding, raising, and maintaining a variety of goldfish species. Proven expertise in creating optimal aquatic environments to ensure the health and vitality of fish. Committed to providing high-quality goldfish to hobbyists and retailers.
Background	Aquatown Goldfish Farm, Aquatown, FL - Goldfish Breeder (January 2012 - Present): Successfully bred multiple species of goldfish, maintained breeding logs, ensured optimal tank conditions, and managed feeding schedules. Consistently achieved high survival rates and vibrant coloration in goldfish. Provided customer education on goldfish care.	*Not Included in JSON*
Qualifications	High School Diploma from Aquatown High School. Certified in Aquaculture Management.	High School Diploma from Aquatown High School. Certified in Aquaculture Management.
Skills	Breeding Techniques, Water Quality Management, Fish Health Monitoring, Customer Service, Tank Maintenance, Aquatic Plant Care, Problem-solving, Detail-oriented.	Breeding Techniques, Water Quality Management, Fish Health Monitoring, Customer Service, Tank Maintenance, Aquatic Plant Care, Problem-solving, Detail-oriented.

Fig. the comparison of the extraction original text vs Json.

The fine tuned model managed to do a good extraction, it totally missed the background information but this is understandable as it was never trained to extract this information.

Here below a second attempt:

Hiroshi Tanaka | 123 Tech Lane, Tokyo, Japan | hiroshi.tanaka@example.com | (123) 456-7890 | **Experienced TV and Radio Repair Technician** with over 15 years of experience in diagnosing, repairing, and maintaining a wide range of television and radio models. Proven ability to troubleshoot complex issues and provide efficient solutions to ensure optimal performance. Dedicated to delivering high-quality service and customer satisfaction. **Working Experience:** Tokyo Electronics Repair Shop - Senior Technician (2009 - Present): Led a team of technicians, handled complex repair tasks, and trained new employees. Consistently achieved high customer satisfaction ratings through efficient and reliable repairs. Specialized in both modern and vintage models, staying updated with the latest technologies and repair techniques. Developed and implemented training programs for new hires, improving overall team performance and efficiency. Nihon Tech Services - Technician (2005 - 2009): Specialized in repairing vintage radios and televisions, provided exceptional customer service. Maintained a high rate of first-time fixes, ensuring minimal repeat visits. Built strong relationships with customers through clear communication and reliable service. **Diplomas:** Diploma in Electronics Repair, Tokyo Technical Institute. **Skills and Expertise:** Advanced Troubleshooting, Component Replacement, Customer Service, Technical Training, Time Management. Proficient in using diagnostic tools and repair equipment, capable of handling both hardware and software issues. Strong attention to detail and problem-solving skills, able to work under pressure and meet tight deadlines. Excellent communication and interpersonal skills, committed to continuous learning and professional development. Fluent in Japanese and English, facilitating effective communication with diverse clients.

The json output:


```
{
  "resumes": [
    {
      "name": "Hiroshi Tanaka",
      "email": "hiroshi.tanaka@example.com",
      "phone": "123-456-7890",
      "experience": "Experienced TV and Radio Repair Technician with over 15 years of experience in diagnosing, repairing, and maintaining a wide range of television and radio models. Proven ability to troubleshoot complex issues and provide efficient solutions to ensure optimal performance. Dedicated to delivering high-quality service and customer satisfaction.",
      "education": "Diploma in Electronics Repair from Tokyo Technical Institute.",
      "skills": [
        "Advanced Troubleshooting",
        "Component Replacement",
        "Customer Service",
        "Technical Training",
        "Time Management",
        "Proficient in Diagnostic Tools",
        "Problem Solving",
        "Communication Skills",
        "Interpersonal Skills",
        "Fluent in Japanese and English"
      ]
```

```
    }
  ]
}
```

And finally the comparison:

Name	Hiroshi Tanaka	Hiroshi Tanaka
Email	hiroshi.tanaka@example.com	hiroshi.tanaka@example.com
Phone	(123) 456-7890	123-456-7890
Experience	Experienced TV and Radio Repair Technician with over 15 years of experience in diagnosing, repairing, and maintaining a wide range of television and radio models. Proven ability to troubleshoot complex issues and provide efficient solutions to ensure optimal performance. Dedicated to delivering high-quality service and customer satisfaction. **Working Experience:** Tokyo Electronics Repair Shop - Senior Technician (2009 - Present): Led a team of technicians, handled complex repair tasks, and trained new employees. Consistently achieved high customer satisfaction ratings through efficient and reliable repairs. Specialized in both modern and vintage models, staying updated with the latest technologies and repair techniques. Developed and implemented training programs for new hires, improving overall team performance and efficiency. Nihon Tech Services - Technician (2005 - 2009): Specialized in repairing vintage radios and televisions, provided exceptional customer service. Maintained a high rate of first-time fixes, ensuring minimal repeat visits. Built strong relationships with customers through clear communication and reliable service.	Experienced TV and Radio Repair Technician with over 15 years of experience in diagnosing, repairing, and maintaining a wide range of television and radio models. Proven ability to troubleshoot complex issues and provide efficient solutions to ensure optimal performance. Dedicated to delivering high-quality service and customer satisfaction.
Education	Diploma in Electronics Repair, Tokyo Technical Institute.	Diploma in Electronics Repair from Tokyo Technical Institute.
Skills	Advanced Troubleshooting, Component Replacement, Customer Service, Technical Training, Time Management. Proficient in using diagnostic tools and repair equipment, capable of handling both hardware and software issues. Strong attention to detail and problem-solving skills, able to work under pressure and meet tight deadlines. Excellent communication and interpersonal skills, committed to continuous learning and professional development. Fluent in Japanese and English, facilitating effective communication with diverse client	Advanced Troubleshooting, Component Replacement, Customer Service, Technical Training, Time Management, Proficient in Diagnostic Tools, Problem Solving, Communication Skills, Interpersonal Skills, Fluent in Japanese and English

Fig. the comparison of the extraction original text vs Json.

The conclusion is that the model does a good job at extracting information, even when exposed to jobs that it has never seen before. This exemplifies the language capabilities of LLMs, in a narrow knowledge context. Without this technology HR departments are forced to go sift through CV manually or use keyword matching techniques that might miss relevant experience or skill if the candidate did not use the exact keyword.

I, **Gabriele Monti**, declare that the work carried out and reported on by this document and the contents of this document, titled **Lightweight Fine-Tuning of Large Language Models To Be Used In Cloud Applications in Document Manipulation**, and accompanying artefacts, except for where attributed to other sources by way of citations within the body of this document and listed under the References section of this document, is formed of my own original work, which has never before been produced for any other purpose, and has been created solely as part of delivery of **Data Science** for or under supervision of **Taolue Chen**.
All information in this document and accompanying artefacts has been obtained and presented in accordance with the relevant ethical conduct and academic rules.

Signature:

Date:
14/09/24

DIGITAL RECEIPT

This receipt acknowledges that Turnitin received your paper. Below you will find the receipt information regarding your submission.

Submission Author	GABRIELE MONTI
Turnitin Paper ID (Ref. ID)	239553023
Submission Title	Lightweight Fine-Tuning of Large Language Models
Assignment Title	Project report submission 18/09/24 - 14:00 (Word/PDF only)
Submission Date	14/09/24, 10:30

Print