Contents

1 Introduction

1.1 Conclusions of exploratory data analysis

1.2 Large Language Models

LLMs [1] are a type of model architecture that both takes input and generates output in natural language. They can take in unstructured data in prose, or any sort of structured data that can be expressed somehow in prose - keeping the structure or not. They are built via a training process, where they "read" a typically massive corpus of existing text in order to learn the patterns that natural language is built upon. Then, the trained model can be accessed in a stage called inference, where the model applies what it's learnt to a potentially new piece of prose. The model continues the piece of prose, repeatedly predicting what word is most likely to come next[1]. This behaviour can be adapted to create a chat interface which can be seen in many places online - for example, with ChatGPT.

[1]The input to the LLM, which it adds text onto, is called the prompt. When adapted to be in a chat interface, we refer to the input, for example a question, as the prompt and the output of the model is called the response.

1.2.1 Relevance to the task

LLMs are uniquely suited to recommendation tasks.

Firstly, they excel at pattern identification. This extends to patterns in people's interests and purchasing habits, even if those patterns are complex. If a user has already purchased a belt, a basic model may recommend another belt - whereas an LLM could understand that since a belt has been purchased already, the customer is more likely to prefer a pair of trousers to accompany the belt. This can be seen as implicit feature engineering, since there is no required manual work to find these patterns ourselves. The LLM is relied upon to find these relationships.

Next, LLMs can handle natural language and unstructured data. When describing a product, information does not need to fit into any predefined category - if one product innately has different properties from another (for example, the buckle type of a belt not fitting into the properties of a pair of trousers), the extra information can be provided to the LLM without changing the structure of other products' descriptions. This is in stark contrast to traditional machine learning architectures.

LLMs handle missing data well [2] - if we don't yet know the age or gender of a customer but do have their purchase history, we can nevertheless make a prediction. The natural language input permits us to leave parts blank, or to state the lack of knowledge with a word like 'null' or 'unknown'. While this prediction would assuredly be of a lower quality, this is in stark contradiction to other architectures which would suffer from a significantly worse drop in quality, or would not be able to make a prediction at all.

Lastly, LLMs can address the cold-start problem in new ways. The cold-start problem is one specific to recommendation systems - how do you recommend a product to a person with entirely new characteristics, for example if they're shopping from a country with no data so far? An LLM would be able to generalise to similar countries, such as the USA to Canada. Furthermore, the cold-start problem is two-fold - how can a product be recommended if it is a new product and has no previous purchases? Here, a continuously updating LLM would be able to understand the characteristics of the new product, and its typical consumer base. This is a benefit that won't be explored in this paper - continuous, in-production fine-tuning would be required, which is computationally expensive and time-consuming.

1.3 The need for fine-tuning

A model without any fine-tuning process will generate output in prose. This is ideal for creative writing, or for providing the answer to a question, but falls short here. We want the model to output a SKU[2], and nothing else. A model which outputs large amounts of text along with a SKU is computationally inefficient - LLMs are more expensive to run than standard recommendation systems, so the cost must be controlled as much as possible.

In essence, fine-tuning takes a well-trained generalist, and transforms the model into one which is especially skilled at one specific skill that we need it to excel at.

When a base model (one which has not been fine-tuned) is prompted to predict what

[2]SKU: Stock Keeping Unit, a unique identifier assigned to each product sold by the retailer

product a customer is most likely to purchase, it won't have any knowledge of products outside of ones that are in the prompt. This restricts the universe of potential products, and if there are no products in the prompt, then it won't even know the structure of the SKU and so will not be able to give even a single valid prediction.

Firstly, fine-tuning teaches the model the structure of the SKUs, as well as how we expect a response from it. This means we don't have to do any sort of regular expression that plucks the prediction out of a piece of prose - instead the model will output the SKU on its own. Secondly, during the fine-tuning process the model learns the underlying patterns of consumer behaviour present in the model - base models are trained on large corpora of text, but will have no domain-specific knowledge of Burberry's customer's shopping habits. The extent to which this learning process exists is an open question, and one which will be explored later on.

It may be beneficial to train an LLM from scratch on the Burberry data, but we do not explore that for a few reasons. Firstly, training an LLM requires massive amounts of data, in the range of terabytes. The base model we will use was trained on 15 trillion tokens[3], or approximately 60 terabytes of data. The training process also requires millions of GPU hours to complete. Next, the generalist side of an LLM isn't something we want to lose - this general knowledge inside the model will give it a sense of fashion preferences, seasonality and more. Losing this knowledge would hurt performance.

1.3.1 Our base model

A fine-tuning process modifies a base model - here, we use the Llama 3.1 8B Instruct model as our base model, released in July of 2023. The 8B refers to the number of parameters inside the model, 8 billion. This is large enough to give a decent quality of prediction, while small enough to be fine-tunable on the Databricks platform. The 'Instruct' keyword tells us that the model is already fine-tuned by Meta AI for out-of-the-box improved performance on instruction tasks. From here on, we refer to this Llama 3.1 8B Instruct model as the base model.

1.4 Comparing LLMs to traditional recommendation models

Currently, there are several architectures used to create recommendation models.

The most naive is collaborative filtering (CF) [4]. In brief, CF methods follow two approaches; user-based and item-based. A user-based CF model will first find other users similar to the target user, in order to recommend the target user items purchased by those other users. Item-based CF models find items most similar to previously purchased items, in order to recommend those items. However, CF approaches tend to struggle in sparse datasets (few interactions per user or product), such as luxury goods, where purchase frequency is low. Traditional CF approaches also force a choice between looking at similar customer characteristics, or alternatively previously purchased items - you can't have both [5][6]. There are approaches that combine both, but typically require extensive feature engineering - a significant difference compared to LLMs, which require almost no feature engineering at all.

More advanced deep learning models have become more prevelant recently [7] for recommendation models, with technques including Multilayer Perceptrons, Sequential Neural Networks and Recurrent Neural Networks. Deep learning architectures see improvements

over CF approaches in terms of their sequence modelling (for example, implicitly understanding the nature of a purchase history) and their ability to handle more complex relationships. Like LLMs, they require a lot of data for training.

Transformers have previously been used for recommendation models using the BERT4Rec architecture [8], but this is limited in only predicting the next product via previously purchased products. Using an LLM allows for more information to be fed into the model - not only the previously purchased products, but other customer characteristics like their age and gender.

1.5 Downsides to using LLMs

LLMs, during inference, do not update any parameters. This means that any trends prevelant in the training dataset will maintain for as long as that specific model is in use. In the context of fashion, an ever-changing landscape, a clear process would be to be constantly retraining and updating the model. While fine-tuning is a comparatively cheap process compared to initial model training, continuous fine-tuning could become expensive.

Secondly, using only customer and transaction data ignores a large part of the fashion landscape. For example, celebrity endorsements could shift consumer behaviour in a way that the model cannot adjust for. However, this isn't a loss against other existing recommendation model architectures as none of them can account for similar external factors either. Furthermore, LLMs would be most suited to be extended to account for such factors, via an adjusted prompt that included data from social platforms.

Next, inference costs are more significant compared to traditional models. For the business use case, using an LLM-based recommender system must be profitable. At a glance, this is simple - the extra profits from the extra sales must be greater than the total cost of inference. However, this is exceedingly hard to quantify. Even if we recommend to a customer the product they are most likely to purchase next, there is no guarantee that they will purchase it - many customers know what they want before purchasing[3]. Then, the question becomes one of probabilities. If a model has a 90% accuracy when recommending a product with £1 profit margins, but a 10% accuracy when recommending a product with £100 profit margins, the most profitable route would be to overwhelmingly recommend the more profitable product even though it will be purchased less. Inference costs being larger for LLMs will exacerbate this problem, as the cost of getting the recommendation wrong will be seen on the bottom line.

2 Background

Transformers are a type of neural network [10] that builds upon previous neural network architectures. Previous state of the art models were recurrent neural networks (long short-

[3]Studies [9] suggest that 40% of spending in e-commerce is from impulse purchases, a type of spending which recommendation models target. However, this isn't specifically for luxury fashion, which has much higher prices than alternative stores and so it would be reasonable to assume that the true value would be less than 40% in our case given that people would be more likely to purchase a cheaper product when unplanned.

term memory and gated recurrent neural networks), which the transformer improves[4] upon by removing the recurrent nature, and replacing it entirely with the self-attention mechanism.

2.1 Limitations of previous architectures

Recurrent Neural Network based architectures suffered from two major problems [12]. Firstly, tokens were processed sequentially, meaning each token in a sentence was processed one after the other. This made training slow, and didn't allow for the efficiency allowed by modern GPUs. The sequential nature also limited their understanding of context - long sentences would struggle to be understood by the model. The second problem is to do with their backpropagation implementation where gradients are multiplied over many steps during training, leading to either vanishing gradients (gradients so small that the network cannot learn long range dependencies) or exploding gradients (gradients so large that the network becomes unstable).

2.2 The transformer model

A transformer is made up three parts;

- an attention mechanism,
- a normalisation component,
- and a feed-forward network.

Modern LLMs are made up of many transformer blocks, one after each other.

The high level description given earlier, that at it's core a transformer works by predicting what word is most likely to come next, remains mostly true. The following descriptions build up the transformer model in more detail.

2.2.1 Tokens & token embeddings

Firstly, a transformer does not just predict what word is most likely to come next. Transformers have a vocabulary made up of tokens, rather than words. Sometimes these tokens are words, sometimes they are not. The models in Vaswani et al. (2017) had a vocabulary made up of 32,000 tokens for the English-French translation task, whereas our Llama base model has a vocabulary made up of 128,000 tokens [13] handling eight languages.

As an example, take the sentence: "Using tokenisation, we convert text into smaller subwords for the model to process", which could be tokenised as: 'Using', '_token', 'isation', ',', '_we', '_convert', '_text', '_into', '_small', 'er', '_sub', 'words', '_for', '_the', '_model', '_to', '_process', where the _ represents a space before the word. Tokenisation makes the model more efficient since it doesn't need to understand the meaning of every possible suffix and prefix of a word. Furthermore, it allows the model to generalise to unseen words, by breaking them down into what it can work out as root words, prefixes and suffixes.

[4]When we say a model 'improves', we are referring to it's performance improving on common benchmarks such as MMLU [11].

Given the token vocabulary, each token is expressed as a vector - called the embedding. Early approaches used static embeddings from a technique such as Word2vec [14]. Modern approaches treat these embeddings as something else that can be learned by the transformer. Firstly, using Byte-Pair Encoding [15], tokens are continuously assigned based on the most popular patterns in the training corpus, until there are as many tokens as the vocabulary size allows for. This process minimises the average amount of tokens to cover the average piece of text, since more popular but longer patterns are prioritised for token allocation over shorter but less popular patterns. Once the tokens have been defined, embeddings are created, a vector of length $D = 4096$ for our base model. Initially, these are randomly generated, with values drawn from a Gaussian distribution.

During the training process, tokens are adjusted to reflect the semantic meaning of each token. The motivation behind having a high embedding dimension D is to let the model learn meaning in as many different ways as possible. For example, for one dimension to express gender. We can see that with the following diagram from [16], a quick explanation of work done in this paper [17].

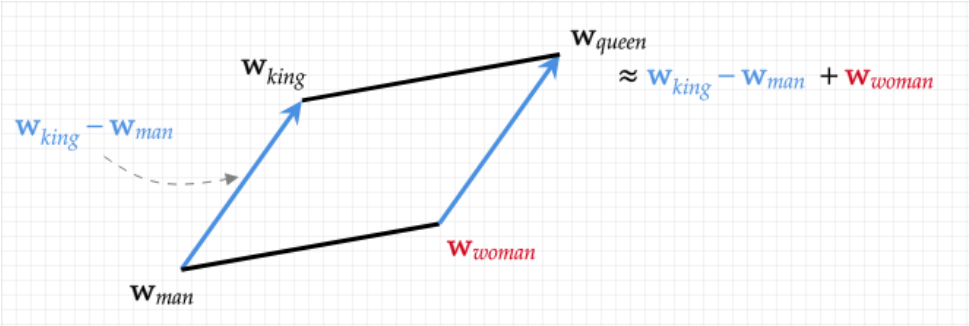

Figure 1: Explaining embeddings

Here we see a 'slice' of the D-dimensional embedding space. The four labels W_{king}, W_{queen}, W_{man} and W_{woman} show how the learned token embeddings for the words could differ. We see the vector of $W_{king} - W_{man} \approx W_{queen} - W_{woman}$. This is like saying, "man is to king how woman is to queen". These token embeddings are all stored in the embedding matrix, a $V \times D$ matrix, for token vocabulary size V.

Skipping over how the parameters (the actual numbers) in the embedding matrix are found, we can discuss positional encoding. Positional encoding is how the model understands which word comes after which - how it knows that "the wolf chases the sheep" and "the sheep chases the wolf" are two different sentences. In other neural network architectures like recurrent or convolutional neural networks, the recurring or convoluting nature of the network means this is unnecessary. Transformers have no such aspect, so there is a different mechanism to allow the model to understand the positions tokens take in sentences.

2.2.2 Positional encoding

Positional encoding occurs once the input has been tokenised, but before it is passed into the transformer layers. Vaswani et al. (2017) used relative positional encoding. First map

each position in the in the input sequence to a D-dimensional vector [5], call it PE, then add this vector onto the token embeddings before passing it into the transformer.

In this implementation of positional encoding, sinusoidal waves are used to capture the dependencies that words in a sentence have amongst each other. For each token at position p and each embedding dimension i, we calculate the PE matrix as

$$PE_{(p,2i)} = \sin\left(\frac{p}{10000^{2i/D}}\right),$$
$$PE_{(p,2i+1)} = \cos\left(\frac{p}{10000^{2i/D}}\right).$$

Then the final input to the transformer is the sum

$$\text{Final Input} = \text{Token Embedding} + PE.$$

This isn't a perfect solution, and the Llama models find improvements by using rotary positional encoding [18] as a method to improve the model. A brief explanation on this technique follows.

2.2.3 Attention

The attention mechanism [19], and more specifically self-attention, is a key breakthrough made by the transformer architecture. Instead of only using token embeddings to convey meaning, attention allows the transformer to understand the semantic meaning of each word in a sentence. For example, the word bright can have multiple meanings, referring to light levels or smarts. It's very clear to the human reader that the word "bright" has two different meanings - nevertheless, both would fall under the same token, with the same embedding.

This is our motivation for attention, a process which allows the transformer to understand the semantic meaning of each word in the sentence, rather than each word on it's own. It applies transformations to the embedding of each token, based on the words and the positions of the words around it.

Terminology: Given two tokens A and B, if A 'attends to' B, some of the semantic meaning of A is imbued in B. For example in the sentence 'the lazy dog', 'lazy' could attend to 'dog', such that the embedding of 'dog' would be transformed in such a way that the model would understand it to be more lazy. Keys, queries and values: at a high level, queries can be seen as questions that one token asks other tokens. Keys are how the other tokens say they have a relevant answer, and values are the answers given if the keys and queries match.

In practice, our base model has 32 transformer layers, and 32 heads of attention (this is called multi-headed attention). Different heads of attention can be parallelised, such that if we understand one layer, we can conceptually understand all the other heads operating in parallel at each layer. Self-attention is called as such because it operates entirely within one sequence of tokens, with tokens attending to each other. Other architectures may use cross-attention [20], which is more applicable for translation and so not discussed here.

[5] The positional encoding vector needs to be D-dimensional because it has to be added to the embedding vectors. If they aren't the same length, we can't do element-wise addition.

Self-attention takes an embedding matrix X of shape $n \times D$, for an input sequence of length n. Initially, this embedding matrix will just be the concatenated input sequence $[x_1, x_2, ..., x_n]$, where $x_i \in \mathbb{R}^D$ is the raw input token cast to it's embedding vector. Recall that D represents the dimension of each embedding vector, 4096 for our base model.

To calculate queries Q, keys K and values V, we apply three linear transformations to the input X

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V,$$

where each of W_Q, W_K, W_V are learned during the training phase, and are of shape $D \times d_k$, where d_k is typically set to D/h, for h the number of heads at each layer. For our base model, we find $d_k = 4096/32 = 128$.

Each token now has it's own query, key and value vector that define how it interacts with other tokens[6].

Now calculate attention scores S, which determine how much focus each token should give to others. This is done by computing the dot product of queries and keys,

$$S = QK^T.$$

Geometrically, this makes sense given our intuition of queries as representing what a token is searching for, and keys as representing what information is available in another token. Then the dot product measures alignment between the two vectors - so a larger dot product would mean the directions are more similar, meaning the query and key match better.

Then scale S for numerical stability at the next stage, giving

$$S_{scaled} = \frac{S}{\sqrt{d_k}} = \frac{QK^T}{\sqrt{d_k}}.$$

Here see that $S_{scaled_{ij}}$ must measure how how relevant token j is to token i - i.e., how much token j should attend to token i. For brevity of notation, we refer to S_{scaled} as S from here.

We want to get a probability distribution for the scores, but S will have arbitrary magnitude - so we apply the softmax function to normalise the attention scores into probabilities. Below is a graph of the softmax function - note it's sigmoid-like shape.

[6]Our base model uses a slightly modified version of this to significantly reduce inference memory overhead by grouping keys and queries, using a technique known as grouped-query attention [21]. The overall intuition and theory remains the same, so we won't discuss it further.

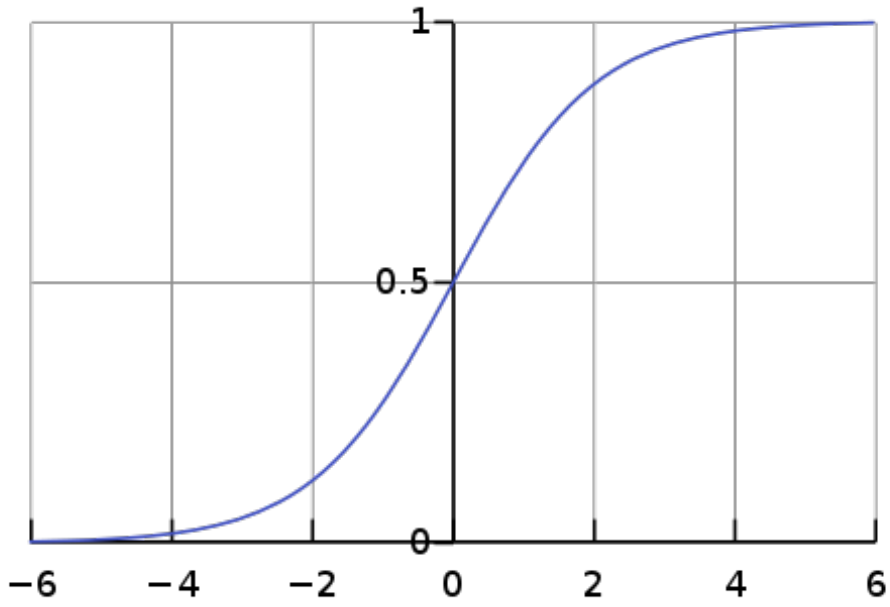

Figure 2: Softmax Function

We compute the attention matrix A using

$$A_{ij} = \text{softmax}(S_{ij}) = \frac{\exp(S_{ij})}{\sum_{k=1}^{n} \exp(S_{ij})},$$

recalling that n represents the length of the input sequence.[7]

This normalises the attention scores so they form a probability distribution summing to 1 over each row, i.e.,

$$\sum_{j=1}^{n} A_{ij} = 1.$$

Now, the attention matrix represents a normalised value of how much token j should attend to token i.

A key point here for the generative nature of modern LLMs is that we don't want later tokens to attend to earlier tokens - this is crucial for the auto-regressive nature. To achieve this, we set every value in S below the diagonal to be $-\infty$, such that they receive an attention score of 0 once softmax is applied. We could apply this after the softmax, but then the columns wouldn't sum to 0, therefore wouldn't form a probability distribution. Setting these values to 0 is a process called masked self-attention.

There are some alternatives to using softmax, but both fall short. We could use a sigmoid function, but this wouldn't result in a probability distribution. Or, we could use a hard-max, an argmax like solution. However, this wouldn't be differentiable which is necessary for the gradient descent techniques used in the learning process discussed later.

[7]Llama models leverage modern GPUs effectively by calculating the attention matrix using FlashAttention [22], which describes itself as an IO-aware method. It uses tiling and recomputation avoidance to reduce the count of memory read/writes. The exact implementation of this is more of a hardware discussion, so will not be discussed further.

Now that we have the attention matrix, we can incorporate the value matrix V to find the output matrix O, by doing

$$O = AV.$$

We can think of the attention matrix at this point as weighting the value matrix, so that the meaning it adds to token i from token j is respective of how strong token j attends to token i.

At this stage, O represents each token along with extra context added by the attention mechanism. From here, we want to pass it through one more transformation, such that the output matrix is the same dimension as the input matrix, and compatible with the next layer. This uses another learned weight matrix, the projection matrix, W_O, by

$$O' = OW_O,$$

where W_O has shape $d_k \times D$, the same dimensions as the input. Aside from getting back into the same space as the input, the re-projection into the higher D-dimensional space allows more complex ideas to be added to the input embedding matrix.

So far, we've discussed a single head of attention, and up to now can imagine that all the h heads have been operating in parallel. It's at this point where they all come together. First of all, it's important to state that each head has it's own query, key and value matrices. The output matrix, W_O, is the same between each.

For each head $i \in \{1, 2, ..., h\}$, we will therefore have $O^{(i)} = A^{(i)}V^{(i)}$, each being the output, attention and value matrices computed by each head. We now concatenate them to produce the final output matrix of that attention layer, doing

$$O = \text{concat}(O^{(1)}, O^{(2)}, ..., O^{(h)}).$$

In practice, this is also where we'd do the multiplication by the projection matrix W_O (rather than before combining the heads), i.e.,

$$O' = \text{concat}(O^{(1)}, O^{(2)}, ..., O^{(h)})W_O.$$

Going back to multi-headed attention, the motivation for using more than one head is to capture different aspects of the input embedding matrix simultaneously. For example, take the prompt "The scientist walked into the laboratory and picked up the...". We could imagine one head focusing on the last few words, determining that it's most likely to be a noun next. Another could focus on more longer range dependencies, adding on that whatever comes next should be scientific.

Finally, we add the output matrix O' onto the input matrix X, effectively adding on the context found. This gives us $\tilde{O}$ as so:

$$\tilde{O} = O' + X.$$

We refer to $\tilde{O}$ as the residual connection output, because it adds in a residual connection O' onto the original input matrix. This technique was introduced in [23] for computer vision, but have become standard for transformers. It brings several gains over passing on just the output matrix, for several reasons. One significant benefit is that it improves

gradient flow during training - if a transformer layer isn't learning anything useful, it lets future layers just fall back on the input matrix X.

Rotary positional encoding

Rotary positional encoding (RoPE) achieves the positional encoding goals by rotating the key and query vectors for each token in such a way that if two tokens are the same distance away from each other in different contexts, they will be rotated by the same value in both scenarios. Rather than adding on fixed positional encodings, the positional information is added within the attention mechanism itself. This also allows the model to generalise to longer sequences. One of the larger arguments for why RoPE improves upon previous methods is that it "helps to decay attention coefficients as the relative distance grows"[24], meaning that on average as the distance between tokens grows, the attention from the earlier to the later token will decrease - which is what we'd expect from natural language.

However, recent work in mechanistic interpretability finds evidence contrary to these claims [24], suggesting that we don't really understand how RoPE finds the empirical model improvements that it does - and potentially that there could be better ways of performing the positional encoding.

2.2.4 Normalisation

During the training process, models tend to see internal covariate shift - the process whereby changing parameters via backpropagation causes the distribution of the output embedding matrices at each step will affect subsequent layers, as the meaning of their inputs would be different to what they'd 'expect'. Some transformer architectures (like the original transformers paper, Vaswani et al. (2017)) solve this using layer normalisation (LayerNorm) [25], which "can substantially reduce the training time".

LayerNorm here would be normalising the output matrix, but the Llama family achieves the same goals in a slightly different way. Rather than operating on the residual connection output after the attention mechanism, Llama models use RMSNorm [26], which instead operates before the attention mechansim.

Pre-normalisation, and the use of RMSNorm, allows error gradient to flow directly throw each transfomer layer, without having to go through the normalisation processes. Empirically, significant training speed increases are found with this pre-normalisation set-up [27].

LayerNorm achieves it's goals in two ways:

- "re-centering invariance, which makes the model insensitive to shift noises in inputs and weights."
- "re-scaling invariance, which preserves output representations when inputs and weights are randomly scaled."[28]

The RMSNorm [26] authors find that by only doing the re-scaling of the invariance, little is lost, and that RMSNorm is "similarly or more effective" than LayerNorm.

Given each token embedding x_i, the RNSNorm equation is:

$$\text{RMSNorm}(x_i) = \gamma \odot \frac{x_i}{\sqrt{\frac{1}{D}\sum_{j=1}^{D} x_{ij}^2 + \varepsilon}},$$

where γ is a learnable parameter. Note that $\odot$ denotes the Hadamard product (element-wise matrix multiplication).

This normalisation achieves the goal of avoiding the problem of internal covariate shift, avoiding unnecessary computation from LayerNorm.

2.2.5 Feed-forward networks

Once the attention process has enriched the embedding matrix with context from other tokens, the feed-forward network adds a non-linear aspect. This is motivated simply by the knowledge that speech and ideas are fundamentally complex and non-linear.

The feed-forward network (FFN) is the last part of the transformer block. It consists of three parts;

- a linear transformation,
- a non-linear activation function,
- another linear transformation.

We can take a rough understanding of how a feed-forward network works by thinking of single hidden layer in a multi-layer perceptrons (MLP), with a few important differences. While both use a similar activation function and have very simlar mathematical structure, an MLP operates on all inputs at once, compared to the FFN in which each token is processed independently - there's no mixing of information between tokens.

Significant literature has been written on the idea that how we train and run inference on perceptron models, and neural networks in general, may actually be how our own neurons operate [29][30][31]. This is encouraging! Our intuition about how these models work, and how to improve them, could very well be accurate to how our brains work, and how we really think ourselves.

The FFNs used by the Llama family are slightly different to vanilla transformer models. Llama models use a gated variant of the FFN known as SwiGLU [32], with the motivation that multiplicative interactions (via the gate) can improve expressiveness and depth of ideas more than just summative effects. Empirically, these gates are found to improve performance more than just increasing the number of transformer blocks [32].

For each vector $x \in R^D$ of the embedding matrix, the FFN performs the following.

1. Firstly, two linear projections of x into a higher dimensional space d_{ff}, in Llama $d_{ff} = 4D$

   $$h_{act} = xW_{act} + b_{act}, \quad h_{gate} = xW_{gate} + b_{gate}, \quad \text{where } h_{act}, h_{gate} \in \mathbb{R}^{d_{ff}}.$$

   This projection into a higher dimensional space also allows the model to understand more complex ideas, and the model is made more effective as a result. This projection was made a standard by Vaswani et al. (2017).

2. Now we use the gating mechanic, doing

$$\tilde{h} = \sigma(h_{act}) \odot h_{gate}, \quad \text{where } \sigma \text{ is the SiLU activation function.}$$

3. Having achieved our goal of non-linearity, we now want to project the $\tilde{h}$ matrix back down to the original dimension, doing

$$\text{FFN}(x) = \tilde{h}W_2 + b_2.$$

Note that W_{act}, W_{gate}, W_2 are all learned weighting matrices, and b_{act}, b_{gate}, b_2 are learned bias vectors. They are unique to each block, but each token in the block passes through the same ones.

In step 2, the use of the SiLU function (and the gating mechanism) is a key difference between the Llama family and other models. In the original transformers paper, they use the ReLU function, where $\text{ReLU}(x) = \max(0, x)$. In contrast,

$$\text{SiLU} = x \cdot \text{sigmoid}(x) = x \left(\frac{1}{1 + e^{-x}} \right).$$

Plotting the two activation functions, we can see that they are very similar.

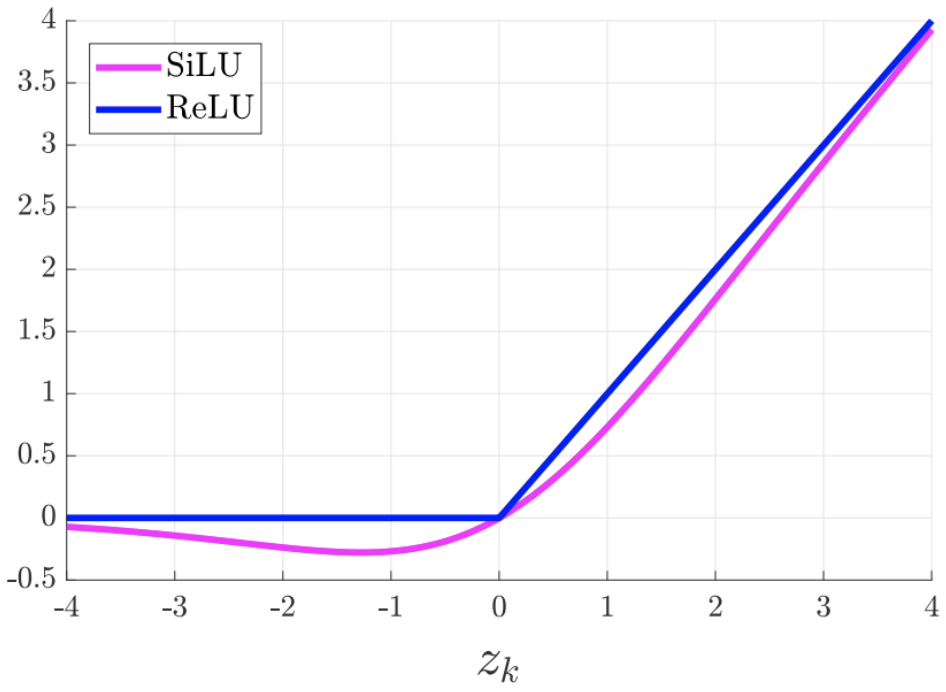

Figure 3: SiLU vs ReLU

From this comparison, we can see how ReLU has a kink at 0, where the SiLU function is smooth, and differentiable throughout. It's thought that this lack of kink is specifically what leads to improvements in effectiveness over ReLU, since ReLU neurons can sometimes get 'stuck' at 0 and never change, in what's known as the Dying ReLU problem [33][34].

2.3 Auto-regressive token generation

Transformer blocks can be chained together, with the output of each FFN layer flowing into the attention layer of the next block repeatedly. Our base model has 32 blocks [13].

How the next token is generated involves a cast back to the token vocabulary space, before sampling from a probability distribution.

Once the embedding matrix has passed through all $L = 32$ transformer blocks, we obtain a matrix $H \in \mathbb{R}^{n \times D}$, where each row $h_i \in \mathbb{R}^D$ is the enriched embedding for token i. We use H for the word hidden, as up until this point, how exactly the tokens are enriched with context is hidden in both the input and output. To convert these hidden states into a probability distribution over the next token n, we project h_n (i.e., the vector containing all the information about the n'th token) into a vector z_n of length V vocabulary size. This means:

$$z_n = h_n W_{voc} \quad \in \mathbb{R}^V,$$

where W_{voc} is another learned weight matrix. Like previous steps, we need to now normalise the z_n vector (called the logits vector) into a probability distribution over all V possible tokens, as so:

$$\mathbb{P}(x_{n+1} = j | h_n) = \text{softmax}(z_n)_j = \frac{\exp\left((z_n)_j\right)}{\sum_{k=1}^{V} \exp\left((z_n)_k\right)}$$

Given this probability distribution, there are two dominant approaches to predicting the next token.

1. **Greedy decoding** - pick the token with the highest probability.
2. **Sampling-based decoding** - draw a token from the probability distribution, respecting the calculated weights.

Sampling-based decoding is useful if you want some creativity, for example in image generation. It allows for more variety and creativity.

For our case however, we will use greedy decoding. We always want to pick the top token, which would be the SKU it calculates to have the highest probability - the product which is most likely to be purchased. If we wanted $k > 1$ predictions, we could take the k tokens with the highest probability.

Once we've taken the next token, we can compute the token at index $n + 1, n + 2$ by feeding the whole sequence back into the model and rerunning it. Modern models like our base Llama models improve on efficiency by caching keys and values [35], to save on recomputing of all matrices at each step.

2.4 Backpropagation

Training an LLM uses backpropagation, a method which has been used to train deep neural networks long before transformers [36]. Backpropagation requires a training dataset, here a large corpus of text, to set the values of all of the weight matrices and bias vectors seen so far. This includes in the attention layers and FFNs, and others that don't fit into either of these such as the W_{voc} we just saw in the token generation stage.

Backpropagation consists of two passes, the forward pass and backward pass. Initially, parameters in the weight matrices are set randomly, taken from either a Xavier distribution [37] or a Kaiming distribution [23]. All bias vectors are set to 0, in order to prevent initialising the model with incorrect bias in early training stages. In the normalisation components, scale parameters are set to 1.

A requirement of backpropagation in any network is a loss function, for which transformers use cross-entropy loss [38]. It's a measure of how different two probability distributions are.

Given a predicted probability distribution $\hat{y}$, where $\hat{y}_{i,j}$ encodes the predicted probability of token i at position j, and y the one-hot encoded true distribution for token i (one for the correct token, 0 otherwise), we compute the cross entropy loss $\mathcal{L}$ as

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{V} y_{i,j} \log(\hat{y}_{i,j}),$$

with N the number of training samples in the batch and V remaining the vocabulary size.

Once we have the cross entropy loss, we want to perform gradient descent. We can imagine the loss function as a hyperplane in a high dimensional space, which is our motivation for moving 'downhill', such that after many iterations we find ourselves in a local minimum. Finding a global minimum is a much harder problem, but not an important one in the context of transformers as the local minimums will tend to exist in a band of similar values [39], so finding a local minimum will be almost just as good as finding a global minimum[8].

To compute our gradient descent, we take the cross-entropy loss for one token on its own:

$$\mathcal{L} = -\sum_{j=1}^{V} y_j \log(\hat{y}_j),$$

again where y_j is the one-hot encoded ground truth, so only that term is non-zero.

We want to compute the gradient of the loss with respect to the logits z_j, i.e., $\frac{\partial \hat{y}_j}{\partial z_k}$. Recalling that the token probability distribution $\hat{y}_j$ is found via softmax'ing the raw logits, we need first to differentiate the softmax function, finding

$$\frac{\partial \hat{y}_j}{\partial z_k} = \begin{cases} \hat{y}_j(1 - \hat{y}_j) & \text{if } j = k, \\ -\hat{y}_j \hat{y}_k & \text{if j} \neq k. \end{cases}$$

Now we differentiate the cross-entropy loss, finding very neatly that

$$\frac{\partial \mathcal{L}}{\partial z_j} = \hat{y}_j - y_j.$$

This makes things very simple - the gradient of the loss with respect to the logits is just the difference between the softmax output and one-hot encoded ground truth.

[8]The cited paper also finds that a global minimum leads to over-fitting of training data, so a local minimum could in fact be preferable.

Backpropagation then involves going backwards through all the transformer layers. We use the chain rule to find that

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial z_j} \cdot \frac{\partial z_j}{\partial W}.$$

Given the gradients for each weight matrix W with respect to loss, we can find the steepest direction at the current point in the hyperplane by finding the vector with the largest gradient. We then adjust the weights so that we move in the opposite direction to that steepest direction, taking us in the direction with the most gradient descent (hence the name of the algorithm).

As to the exact updates to the weight matrices at each step is beyond the requirements of this section. The Llama models use the AdamW optimiser [40], which is the standard optimiser for large transformer models. It incorporates adaptive learning rates per parameter, helping to train large models efficiently, as well as taking ideas from momentum in stochastic gradient descent.

Llama models also leverage stochastic optimisation techniques[9] to avoid calculating the loss function over the entire training dataset - the intuition comes from the central limit theorem, in that the loss function over a small subset of the training dataset will be close enough to the one over the entire dataset. As a result, the training process takes more steps to converge, but each step is significantly faster.

2.5 Fine-tuning

Given a pre-trained transformer model, we are motivated to improve at one specific skill. In our case, this is to predict the product someone is most likely to purchase. Methods involve either adjusting existing weights, or inserting and training new layers. By doing this, we may expect it's performance to degrade in other benchmarks, however for significantly improved performance at our specified task that is an acceptable trade-off.

2.5.1 Methodology overview

Early research [41] proved that fine-tuning of language models was indeed possible. This is now known as full fine-tuning, in which all model parameters are updated using task specific data. This had several challenges.

- Firstly, it was very computationally intensive [41] as all parameters had to be updated. For us, that would involve computing the entire training process over all 8 billion parameters in our model.
- It also came with risk of catastrophic forgetting [42], in which basic general knowledge or even sentence structure understanding could be lost during the fine-tuning process.

Following full fine-tuning, research was done on parameter-efficient fine-tuning [43], aiming to achieve similar results without having to retrain all parameters. This worked by inserting small trainable layers, called adapter layers, in between frozen (not updated) transformer layers. This yielded good performance, with significantly less computational

[9]Note this does **not** mean stochastic gradient descent. The AdamW optimiser is an alternative, not an add-on to SGD. The similarity stems from a core idea of SGD, computing the gradient over a small batch of training data, which is used in both methods.

cost in time and resources. Parameters in the adapter layers made up only 3-5% as many parameters in the full model, but there was still motivation to reduce this further.

2.5.2 Low-Rank Adaptation

Low-Rank Adaptation [44] (LoRA) fine-tuning is a parameter-efficient fine-tuning method, which significantly reduces memory and computation requirements, without losing significant performance over other methods.

The key concept is that updates to weight matrices during fine-tuning processes tend to be low rank - the changes to the matrices lie in a lower-dimensional space than the weight matrices themselves [44]. So rather than update the entire matrix (which is an expensive operation), the updates can be well-approximated using lower-rank transformations.

Given a pre-trained weight matrix $W_{pretrained} \in \mathbb{R}^{d \times k}$, LoRA updates it by

$$W = W_{pretrained} + \Delta W,$$

LoRA transforms the ΔW update to be two individual matrices A and B, such that

$$\Delta W = AB,$$

where $A \in \mathbb{R}^{d \times r}$ and $B \in \mathbb{R}^{r \times k}$, with the rank $r \ll \min(d, k)$. So in the fine-tuning process,

$$W = W_{pretrained} + \Delta W = W_{pretrained} + AB.$$

The original matrix $W_{pretrained}$ is frozen and unchanged by the fine-tuning process. The A, B matrices are learnt via backpropagation over the fine-tuning training data. An appropriate choice of r will significantly reduce the number of trainable parameters, reducing memory usage and improving training speed.

The LoRA process can be applied to any weight matrices in the model - typically, it will be applied to the attention layers (particularly queries and values)[44].

QLoRA

During the investigation, we are motivated to increase the token vocabulary in order to handle each of the SKUs as its own token. By doing this, the size of the model is increased such that it no longer fits in the CUDA memory of a single Nvidia A100 GPU. To reduce the size of the model, we use Quantised Low-Rank Adaptation (QLoRA) [45] to quantise matrices of floating point numbers in the LoRA fine-tuning architecture down to 4 bits. The QLoRA authors find this results in minimal performance losses for their application.

3 Investigation

The aim of our fine-tuning investigation is two-fold.

1. **Feasibility** - can LLMs predict what product someone's going to purchase next? We'll answer this by comparing to rudimentary recommendation models as well as models currently used in industry.

2. **Evidence of learning** - do LLMs learn the underlying relationships powering customer decision making? We'll answer this by looking for evidence of relationships we found ourselves, in several aspects. For example, if we find a product bought predominantly in winter months, does the model predict it predominantly over winter months? If a model proves to have learned relationships that we have found in earlier research, we can hypothesise that it will have learned relationships that we did not manage to find - thus making the case for LLMs as recommendation models.

Our investigation will try to answer the first question by repeatedly iterating on the model in order to create the best model possible. As we go, we will analyse the predictions made by the model in order to answer the second question.

References

[1] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," 2024. [Online]. Available: https://arxiv.org/abs/2307.06435

[2] Z. Ding, J. Tian, Z. Wang, J. Zhao, and S. Li, "Data imputation using large language model to accelerate recommendation system," 2024. [Online]. Available: https://arxiv.org/abs/2407.10078

[3] M. AI, "Meta llama 3.1: The next generation of open models," 2025, accessed: 7 Feb. 2025. [Online]. Available: https://ai.meta.com/blog/meta-llama-3-1/

[4] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 271–280. [Online]. Available: https://doi.org/10.1145/1242572.1242610

[5] X. Su and T. M. Khoshgoftaar, "A survey of collaborative filtering techniques," *Advances in Artificial Intelligence*, vol. 2009, no. 1, p. 421425, 2009. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2009/421425

[6] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, *Recommender Systems Handbook*. Springer, 2011. [Online]. Available: https://link.springer.com/book/10.1007/978-0-387-85820-3

[7] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Computing Surveys*, vol. 52, no. 1, p. 1–38, Feb. 2019. [Online]. Available: http://dx.doi.org/10.1145/3285029

[8] F. Sun, J. Liu, J. Wu, C. Pei, X. Lin, W. Ou, and P. Jiang, "Bert4rec: Sequential recommendation with bidirectional encoder representations from transformer," 2019. [Online]. Available: https://arxiv.org/abs/1904.06690

[9] K. Saleh. (2025) The state of impulse buying (statistics & trends 2025). Accessed: 7 Feb 2025. [Online]. Available: https://www.invespcro.com/blog/impulse-buying/

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[11] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring massive multitask language understanding," 2021. [Online]. Available: https://arxiv.org/abs/2009.03300

[12] N. Zucchet and A. Orvieto, "Recurrent neural networks: vanishing and exploding gradients are not the end of the story," 2024. [Online]. Available: https://arxiv.org/abs/2405.21064

[13] Llama Team, AI @ Meta, "The llama 3 herd of models," 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013. [Online]. Available: https://arxiv.org/abs/1301.3781

[15] P. Gage, "A new algorithm for data compression," *C Users J.*, vol. 12, no. 2, p. 23–38, Feb. 1994.

[16] S. o. I. University of Edinburgh, "King - man + woman = queen: the hidden algebraic structure of words," 2025, accessed: 2025-02-17. [Online]. Available: https://informatics.ed.ac.uk/news-events/news/news-archive/king-man-woman-queen-the-hidden-algebraic-struct

[17] C. Allen and T. Hospedales, "Analogies explained: Towards understanding word embeddings," 2019. [Online]. Available: https://arxiv.org/abs/1901.09813

[18] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu, "Roformer: Enhanced transformer with rotary position embedding," 2023. [Online]. Available: https://arxiv.org/abs/2104.09864

[19] D. Soydaner, "Attention mechanism in neural networks: where it comes and where it goes," *Neural Computing and Applications*, vol. 34, no. 16, p. 13371–13385, May 2022. [Online]. Available: http://dx.doi.org/10.1007/s00521-022-07366-3

[20] M. Gheini, X. Ren, and J. May, "Cross-attention is all you need: Adapting pretrained transformers for machine translation," *arXiv preprint arXiv:2104.08771*, 2021. [Online]. Available: https://arxiv.org/abs/2104.08771

[21] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," 2023. [Online]. Available: https://arxiv.org/abs/2305.13245

[22] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," 2022. [Online]. Available: https://arxiv.org/abs/2205.14135

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[24] F. Barbero, A. Vitvitskyi, C. Perivolaropoulos, R. Pascanu, and P. Veličković, "Round and round we go! what makes rotary positional encodings useful?" 2024. [Online]. Available: https://arxiv.org/abs/2410.06205

[25] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016. [Online]. Available: https://arxiv.org/abs/1607.06450

[26] B. Zhang and R. Sennrich, "Root mean square layer normalization," 2019. [Online]. Available: https://arxiv.org/abs/1910.07467

[27] Q. Wang, B. Li, T. Xiao, J. Zhu, C. Li, D. F. Wong, and L. S. Chao, "Learning deep transformer models for machine translation," 2019. [Online]. Available: https://arxiv.org/abs/1906.01787

[28] S. Dhakal, "The llama family of models, model architecture, size, and scaling laws," 2024. [Online]. Available: https://www.icodeformybhasa.com/p/the-llama-family-of-models-model

[29] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[30] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[31] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Washington, DC: Spartan Books, 1962.

[32] N. Shazeer, "Glu variants improve transformer," 2020. [Online]. Available: https://arxiv.org/abs/2002.05202

[33] A. F. Agarap, "Deep learning using rectified linear units (relu)," 2019. [Online]. Available: https://arxiv.org/abs/1803.08375

[34] L. L. Lu Lu, Y. S. Yeonjong Shin, Y. S. Yanhui Su, and G. E. K. George Em Karniadakis, "Dying relu and initialization: Theory and numerical examples," *Communications in Computational Physics*, vol. 28, no. 5, p. 1671–1706, Jan. 2020. [Online]. Available: http://dx.doi.org/10.4208/cicp.OA-2020-0165

[35] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, Nov. 2015. [Online]. Available: https://doi.org/10.1145/2830555

[36] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. [Online]. Available: https://doi.org/10.1038/323533a0

[37] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html

[38] Z. Zhang and M. R. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," 2018. [Online]. Available: https://arxiv.org/abs/1805.07836

[39] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, "The loss surfaces of multilayer networks," 2015. [Online]. Available: https://arxiv.org/abs/1412.0233

[40] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019. [Online]. Available: https://arxiv.org/abs/1711.05101

[41] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," 2018. [Online]. Available: https://arxiv.org/abs/1801.06146

[42] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences*, vol. 114, no. 13, p. 3521–3526, Mar. 2017. [Online]. Available: http://dx.doi.org/10.1073/pnas.1611835114

[43] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. de Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, "Parameter-efficient transfer learning for nlp," 2019. [Online]. Available: https://arxiv.org/abs/1902.00751

[44] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021. [Online]. Available: https://arxiv.org/abs/2106.09685

[45] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," 2023. [Online]. Available: https://arxiv.org/abs/2305.14314