Demystifying Large Language Models: Weights, GPT Architecture, Training, and Math Basics

Large Language Models (LLMs) like GPT-3 and GPT-4 have gained fame for their ability to generate human-like text. But how do they work under the hood? In simple terms, an LLM is a neural network language model with a vast number of **parameters** (values the model learns) trained on massive amounts of text data . These models use a *transformer* architecture (more on that soon) and learn to predict text – essentially learning the probability of the next word given the previous words . In this article, we'll break down the key concepts – **weights and biases** (model parameters), the **GPT architecture**, the **training process**, and the basic **math** involved – in a concise, undergrad-friendly way.

Weights, Biases, and Model Parameters in Neural Networks

In any neural network, including large language models, **weights** and **biases** are the fundamental parameters that get adjusted during training. You can think of the weights as knobs that scale the influence of different inputs, while biases are like offsets that shift the model's output up or down slightly . Together, weights and biases determine how the input data is transformed at each layer of the network, guiding the model's decisions . For example, when an input (say, a token embedding representing a word) enters a neuron, the neuron multiplies that input by a weight and adds a bias to it – this weighted sum is then passed through an activation function to produce an output for the next layer.

Model parameters is just another term for all the weights and biases in the model. Large language models have an astonishing number of these parameters. (Each parameter is essentially a number that the model can tune.) To give a sense of scale, the original GPT-1 model in 2018 had about 117 million parameters, GPT-2 in 2019 grew to around 1.5 billion, and GPT-3 (2020) jumped to **175 billion** parameters . Modern models like GPT-4 are believed to be even larger. These billions of weights and biases are where the model “stores” the information it learns from training data.

During training, the model *learns* the optimal values of weights and biases by gradually adjusting them to minimize errors. Initially, the weights and biases are set to random values. The training algorithm then feeds the model a huge corpus of text and iteratively tweaks the parameters to make the model’s predictions (e.g. the predicted next word) closer to the true text. In practical terms, moving from one layer of the network to the next means performing a linear operation: multiplying the input vector by a weight matrix and adding a bias vector (often followed by a non-linear activation) . The values in those weight matrices and bias vectors get updated little by little as training progresses. In summary, **weights and biases are the numbers that the training process tunes** – they are the *learned* configuration that enables an LLM to generate meaningful language .

The GPT Architecture: Transformers and Attention

GPT stands for *Generative Pre-trained Transformer*. It’s “generative” because it generates text, “pre-trained” because it is first trained on a broad dataset, and “transformer” because it uses the transformer neural network architecture. The Transformer is the key architecture behind modern large language models, and it was introduced in the landmark 2017 paper *“Attention Is All You Need.”* At its core, a transformer is built from repeated layers (sometimes called transformer *blocks*) that contain a mechanism called **self-attention** along with some feed-forward neural network layers.

A **transformer block** typically consists of a self-attention layer plus some additional layers like feed-forward networks, often with residual connections and normalization in between. GPT models are essentially a stack of these transformer blocks (for instance, GPT-1 had 12 such layers stacked sequentially) . Because each block’s input and output have the same dimensions, they can be stacked to form a deep network . The output of the final transformer layer then goes through one more linear layer to produce a probability distribution over the vocabulary for the next word prediction (achieved by a softmax function at the end) .

What is **self-attention**? It’s the transformative idea that allows the model to weigh the relevance of different words in a sequence to each other. In each self-attention layer, the model looks at a given word (or more precisely, that word’s current vector representation) and determines how much attention to pay to every other word in the context. It does this by comparing "query" vectors and "key" vectors (derived from the words via learned weight matrices) through dot products – essentially measuring similarity – to get a set of *attention scores*. These scores are then converted into weights (using a softmax, which turns them into probabilities that sum to 1) . The model uses these weights to take a weighted average of "value" vectors (another set of transformed word vectors), producing a new representation of the word that now encodes information from the other relevant words . In simpler terms, the model learns to focus on the words that are most important for understanding the current word. For example, in a sentence, a pronoun like "it" might attend to the noun it refers to, so that the model knows what *"it"* stands for.

GPT is a *decoder-only* transformer model, which means it uses only the part of the transformer that generates output from input (as opposed to models that have an encoder-decoder structure for tasks like translation). One important aspect of GPT's architecture is that it uses **masked self-attention**. During training, the model is predicting the next word in a sequence, so it should not peek at the actual next word ahead of time. Masked attention means that when computing attention for a given position in the text, the model is only allowed to look at earlier positions (the words to its left) and *not* any later words to its right . This way, the model can be used to generate text step by step, always looking only at the context so far.

All these mechanisms together let a GPT model ingest a sequence of tokens (words or subword pieces) and produce a continuation one token at a time. Each token is first turned into a numeric **embedding** (a vector) that represents that token in a high-dimensional space. The transformer then processes those embeddings through many layers of attention and feed-forward transformations. The final layer outputs a score for every word in the vocabulary, and the softmax turns these into probabilities – e.g. the model might predict the word "world" has 0.8 probability of coming next versus 0.1 for "word", 0.05 for "whale", etc. The model typically picks the highest-probability option (or samples according to the probabilities) as the next word, then repeats the process for the following word, and so on to generate text.

It's impressive that using this architecture, GPT models can capture a lot of linguistic patterns and even world knowledge. The key is that by scaling up the number of layers and the number of parameters (and of course the amount of training data), these models build very rich internal representations of language. In fact, GPT-3's transformer used 96 layers and a **12,288-**

dimensional representation space for tokens, totaling 175 billion parameters . Those design choices (layers, dimensions, etc.) define the model's capacity. Modern LLMs are so large and complex that nobody can interpret all the weights directly – but empirically, we see that *bigger tends to be better* in terms of performance, provided you also have enough data and computation to train the model .

How Large Language Models are Trained

The training process of a large language model is all about learning to predict text by example. The training data is a giant collection of text (for instance, internet articles, books, websites – in GPT-3's case, about 500 billion words worth of text !). The training is **self-supervised**, meaning the model learns from the raw text itself without needing human-labeled answers – each next word in a sentence serves as a training target for the previous words. Formally, the model is trying to learn the conditional probability *P(next word | previous words)* for sequences of words . If the model can accurately predict the next word for lots of sentences, it has essentially learned the structure of the language.

Here's an overview of the training procedure for an LLM like GPT:

1. **Forward Pass:** The model takes a chunk of text as input (say a sequence of tokens). It processes these tokens through its layers (applying all those matrix multiplications and attention operations) to produce an output – which in language modeling is a predicted probability distribution over the next token. For example, after reading "The cat sat on the", the model might output probabilities for various next words ("mat", "sofa", "ground", etc.).
2. **Loss Calculation:** Next, the model's prediction is compared to the actual next word in the training text (maybe the actual next word was "mat."). The difference is quantified by a **loss function**. A very common choice is *cross-entropy loss*, which effectively measures how far off the predicted probability distribution is from reality (the ideal prediction would put 100% probability on the correct word) . If the model gave "mat" a high probability, the loss will be low; if it gave "mat" a low score, the loss will be high. This loss is a single number representing the model's error on that example.
3. **Backward Pass (Backpropagation):** Now the training algorithm figures out how to adjust the model's weights to reduce that error. This is done via **backpropagation**, which is a procedure to compute the *gradient* of the loss

with respect to each weight and bias. In essence, backpropagation works backwards from the output layer, calculating how much each parameter contributed to the error by using the chain rule from calculus . It’s a bit like tracing back through a maze to see which turns led to a dead end – the algorithm finds which weights need to be nudged up or down to make the prediction more accurate next time .

4. **Parameter Update:** Once the gradients (slopes telling us the direction to adjust each parameter) are known, the model updates its weights and biases slightly in the direction that decreases the loss. This typically happens through an optimization algorithm like **gradient descent** (often a specific variant like Adam optimizer). The weights are nudged by an amount proportional to the gradient (and a chosen learning rate). After this update, if we present the same example again, the model should predict “mat” with a higher probability than before, meaning it has learned from its mistake .
5. **Repeat:** The above steps are repeated for billions of examples. The model goes through the entire dataset multiple times (each full pass over the data is called an epoch, though for huge datasets often it may not even see everything more than once). Over time, the weights converge to values that make the predictions align closely with the actual text, i.e., the model has learned the statistical structure of the language in the training data.

It’s worth noting that training these models is an enormous undertaking. Each forward and backward pass involves *huge* matrix operations – for a model as big as GPT-3, a single pass through the network for one batch of text involves **hundreds of billions** of mathematical operations . And this needs to be done for each of billions of words. In fact, OpenAI estimated that training GPT-3 required on the order of 3 × 10^23 (300 billion trillion) floating-point operations in total . This is why such models are trained on specialized hardware (like clusters of GPUs or TPUs) over many weeks or months. The training of GPT-3, for example, took dozens of high-end chips running for months .

After the **pre-training** phase (learning to predict generic text), the model can also go through a **fine-tuning** phase. Fine-tuning means taking the pre-trained model and training it a bit more on a smaller, task-specific dataset. For instance, one might fine-tune a general LLM on some medical texts to adapt it for healthcare applications, or on dialogue data to make it a better conversational agent. Fine-tuning typically uses orders of magnitude less data and compute than the original pre-training, and it slightly adjusts the weights to specialize on the new task . The combination of pre-training (learning general language patterns) and fine-tuning (adapting to specific needs) is a powerful paradigm that has made it feasible to deploy LLMs for all sorts of applications without training each from scratch.

The Math Behind LLMs (in a Nutshell)

You don’t need to dive into all the equations to get the gist of the math that large language models use. Here we’ll outline the main types of mathematics involved and point you to what you could learn more about:

- **Linear Algebra (Matrices and Vectors):** Neural networks heavily use linear algebra. As we saw, each layer’s computation is basically vector–matrix multiplication plus adding a bias vector . The *transformer’s* attention mechanism is also built on linear algebra operations: it uses dot products (a linear algebra operation) to compute similarity between vectors (queries and keys) . Large weight matrices, vector embeddings for words, and operations like matrix multiplication, addition, and concatenation are the bread and butter of an LLM’s computations . Students who want to explore more can look into concepts like matrix multiplication, eigenvectors (for understanding things like principal components), and singular value decomposition – though the basics of multiplying matrices and understanding vectors in high-dimensional space are the most directly relevant.
- **Calculus and Optimization:** The training process relies on calculus, especially differential calculus, to optimize the model’s parameters. Calculus comes into play through the computation of gradients. Using the chain rule, the training algorithm finds how a tiny change in each weight would change the loss (error) . This gradient is essentially a derivative of the loss with respect to the parameter. Armed with those gradients, the optimizer adjusts the weights in the opposite direction of the gradient (to descend into a loss “valley”). This process is known as gradient descent. Understanding the chain rule and the concept of a gradient is key to grasping how backpropagation works. More advanced math in this category includes partial derivatives and concepts from optimization like momentum and adaptive learning rates, but at its heart it’s calculus guiding an iterative improvement process .
- **Probability and Statistics:** Large language models treat language generation as a probabilistic prediction. The output of the model at any time is a probability distribution over possible next words. The model uses the

softmax function to convert raw scores (logits) into probabilities that sum to 1 . Softmax is essentially an exponential-based normalization that makes some numbers large and others small, but in a smooth way (so that the highest score becomes the highest probability, but even lower-scoring options get some probability mass). The **loss function** commonly used, cross-entropy, comes from information theory and statistics – it measures the distance between the probability distribution the model predicted and the ideal distribution (which would put probability 1 on the correct word and 0 on others) . Minimizing cross-entropy is equivalent to maximizing the likelihood of the training data under the model. In addition, concepts like expectation, variance, and statistical significance underpin how we evaluate models and understand their predictions. Students curious about the math here could explore the softmax formula and why cross-entropy is a natural choice for a loss in classification problems.

- **Other Mathematical Tools:** There are other pieces of math that show up in LLMs as well. For example, **logarithms** are implicit in how the softmax and cross-entropy are computed (cross-entropy loss involves a log of the predicted probability of the correct word). **Information theory** concepts (like entropy) inspire some of these metrics. Furthermore, **Discrete math** appears in how text is tokenized (splitting text into tokens and mapping them to integers), and **modular arithmetic** can appear in positional encoding formulas (used to give the model a sense of word order). For those interested in more advanced aspects, areas like *matrix calculus*, *group theory* (for certain symmetry analyses), or *graph theory* (for interpreting attention heads connections) can also be relevant, though they go beyond the basic gist.

In summary, large language models sit at the intersection of computer science and mathematics. They use linear algebra to handle their many weights and the high-dimensional representations of language, calculus to learn from data via gradient-based optimization, and probability to make predictions and quantify uncertainty . The implementation involves a lot of matrix multiplications and nifty engineering, but conceptually, if you understand those three pillars of math, you have a good handle on what's happening under the hood. And if any of these areas pique your interest, diving deeper into them will only enhance your understanding of how models like GPT work.

By understanding weights and biases, the transformer architecture, the training loop, and the underlying math, you get a clearer picture of what large language *models really do. These models basically distill patterns from huge amounts of text into billions of numerical parameters. When you input a prompt, those parameters – those weights and biases – work together (via all the matrix math and attention mechanisms) to produce a coherent continuation. While we skipped the heavy equations, we covered the key ideas and techniques. With this foundation, you can appreciate both the* ***power*** *and the* ***complexity*** *of GPT-like models. And if you're eager to learn more, you now have some pointers on which concepts (from multi-head attention to cross-entropy loss) to explore further!*

📚 Sources

- Korir, K. (2023, October 18). *Mathematical foundations of large language models*. Towards Data Science. https://towardsdatascience.com/mathematical-foundations-of-large-language-models-b4ce668f9c5b
- Lee, T. B., & Trott, S. (2023, March 20). *Large language models, explained with a minimum of math and jargon*. Vox. https://www.vox.com/future-perfect/23642715/chatgpt-large-language-models-explained
- OpenAI. (2020). *Language models are few-shot learners* (Brown, T. B., Mann, B., Ryder, N., et al.). https://arxiv.org/abs/2005.14165
- Penke, C. (2023, April 27). *A mathematician's introduction to transformers and LLMs*. Medium. https://carolinpenke.medium.com/a-mathematicians-introduction-to-transformers-and-llms-6b1021e8f6f6
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). *Attention is all you need*. Advances in Neural Information Processing Systems, 30. https://arxiv.org/abs/1706.03762