Corpus-Conditioned Bias: Training GPT-2 Models on Eastern and Western Philosophical Texts

Mahatav Arora, Vanshika Singla, Bowen Hui

June 18, 2026

1 Introduction

Large language models acquire knowledge from their training data. The text a model encounters during pre-training determines its vocabulary, conceptual scope, and default responses in uncertain situations. Two models with identical architectures and training procedures will generate different outputs if their training corpora originate from distinct traditions, and these differences may not always be reflected in perplexity scores.

In this thesis, we train two GPT-2 models from scratch: one on a corpus of Western philosophy and the other on a corpus of Eastern philosophy. Both models share the same architecture, tokenizer, and hyperparameters; the only variable is the training text. Therefore, any measurable differences in their outputs that cannot be attributed to chance must originate from the corpus.

We halt the process at the end of pre-training, without fine-tuning or any post-training. This approach is chosen because instruction-tuned models reflect the influence of instruction authors as much as their original corpora, making it difficult to attribute specific behaviours to either stage. By stopping after pre-training, we can isolate the effects of the corpus itself.

The primary challenge lies in measurement. Since the two models are trained on disjoint corpora, there is no held-out test set that is equally representative for both, and same-model perplexity is not directly comparable. The main metric employed is cross-perplexity: each model evaluates text generated by the other, and high cross-perplexity values in both directions indicate that the models have learned distinct distributions. Additional metrics include concept-marker frequencies (assessing whether each model utilizes the technical vocabulary of its respective tradition), type-token ratio and repetition (evaluating output fluency and potential model collapse), and the Bhattacharyya overlap of the output distributions. Some metrics yield inconsistent results; Section 4 discusses the rationale for considering both outcomes valid.

Section 2 provides background information. Section 3 details the corpora, the GPT-2 configuration, and the progressive training pipeline. Section 4 presents the results, and Section 5 discusses the successes and limitations of the approach.

2 Background

2.1 Mathematical Definition: Manu

A language model is a system that, given some text, predicts what text is likely to come next. This section makes that idea precise by expressing it in the language of probability, which is the form we will need to train and evaluate one.

2.1.1 Problem in Mathematical Terms: Manu

We frame language modelling as a *next-token prediction* task [BDVJ03, Gol16]. Rather than asking "what is the next sentence?", we ask "what is the next small piece of text given everything that has come before?" Answering the small question repeatedly turns out to be enough to answer the larger one.

A token is a small unit of text drawn from a fixed list called the *vocabulary* [JM23]. Depending on the system, a token might be a whole word, a subword piece (*run*, *##ning*), or even a single character. The vocabulary itself is finite and decided in advance, and we write it as

$$V = \{w_1, w_2, \ldots, w_{|V|}\}, \tag{1}$$

where $|V|$ is the number of tokens in it.

A piece of text is then a *sequence* of tokens. For a sequence of length T, we write

$$\mathbf{x} = (x_1, x_2, \ldots, x_T), \tag{2}$$

with each x_i drawn from the vocabulary,

$$x_i \in V \quad \text{for } i = 1, \ldots, T. \tag{3}$$

A language model assigns a probability to any such sequence: how likely is this particular ordering of tokens to appear in real text? We write this joint probability as

$$P(x_1, x_2, \ldots, x_T). \tag{4}$$

Working directly with the joint probability of an entire sequence is awkward because the number of possible sequences is enormous (Section 2.1.2 makes this concrete). The *chain rule of probability* gives us a way around it by rewriting the joint probability as a product of simpler conditional probabilities:

$$P(x_1, x_2, \ldots, x_T) = \prod_{i=1}^{T} P(x_i \mid x_1, x_2, \ldots, x_{i-1}). \tag{5}$$

So modelling whole sequences and modelling the next token given the previous ones are the same problem, repeated T times. From here on,n we only need a way to compute $P(x_i \mid x_1, \ldots, x_{i-1})$.

2.1.2 Why a Direct Lookup Table Doesn't Work: Manu

The most obvious way to compute these conditional probabilities is to memorize them: store, for every possible context, the probability of every possible next token in a giant table, and look up the answer whenever we need one. This approach fails for two reasons. The table would be too large to store, and even if we could, almost all its entries would be empty.

Suppose our tokens are English words and our vocabulary contains roughly 50,000 of them:

$$|V| = 50{,}000 \tag{6}$$

$$\mathbf{x} = (x_1 = \text{hello},\ x_2 = \text{phone},\ \ldots,\ x_T). \tag{7}$$

The number of possible sequences of length T is

$$|V|^T, \tag{8}$$

since each of the T positions can be filled by any of the $|V|$ tokens. For a fairly short sequence of $T = 20$ tokens, this gives

$$50{,}000^{20} \tag{9}$$

possible sentences. At 8 bytes per stored probability,y the table would need around $8 \times 50{,}000^{20} \approx 8 \times 10^{94}$ bytes, which is far beyond any practical storage capacity.

The sparsity problem is the second issue. A natural way to estimate a probability is to count how often something happens and divide by the total number of observations. For sequenc,es this gives the maximum-likelihood estimate

$$P(x_1, x_2, \ldots, x_T) = \frac{\text{count}(x_1, x_2, \ldots, x_T)}{N}, \tag{10}$$

where N is the total number of sequences in our dataset. Even with a very large dataset, say $N = 10^{12}$ sequences, the vast majority of length-20 sequences will never have appeared in it. For all of those unseen sequences,

$$P(x_1, x_2, \ldots, x_T) = \frac{0}{10^{12}} = 0. \tag{11}$$

The table, therefore, assigns probability zero to almost every sentence a person might actually type, including ordinary English sentences that happen to differ slightly from anything in the training data. This is the *sparsity problem* in language modelling [JM23], which means that a purely lookup-based approach cannot generalize. It can only repeat what it has already seen.

Before neural networks, the standard way to deal with the sparsity problem was to give up on conditioning on the full sequence and instead condition only on the previous token. This gives the *bigram model*, which was the dominant simple approach to language modelling for many years [JM23]. Rather than computing the full conditional $P(x_i \mid x_1, \ldots, x_{i-1})$, a bigram model approximates it by

$$P(x_i \mid x_1, \ldots, x_{i-1}) \approx P(x_i \mid x_{i-1}). \tag{12}$$

In words: the probability of the next token depends only on the single token immediately before it. The probability is estimated by counting how often each pair of tokens appears together in the training data and dividing by how often the first token appears at all:

$$P(x_i \mid x_{i-1}) = \frac{\text{count}(x_{i-1}, x_i)}{\text{count}(x_{i-1})}. \tag{13}$$

A bigram model is also a special case of the more general *n-gram model*, which conditions on the previous $n-1$ tokens [BDVJ03]. A trigram model uses the previous two, a 4-gram the previous three, and so on. We focus on bigrams in this section because they are the simplest case to reason about and because the limitations they expose carry over to all n-gram models.

Bigrams are a real improvement over the full-sequence lookup table from earlier in this section. Instead of $|V|^T$ entries, a bigram model only needs to store probabilities for pairs of tokens. With $|V| = 50{,}000$, that is at most $|V|^2 = 2.5 \times 10^9$ entries, which is large but actually achievable on real hardware. And many of those pairs never need to be stored explicitly: bigrams that never appear together in training (such as "purple sneeze") can be treated as having very low probability. In contrast, the bigrams the model does see (*the dog*, *the cat*, *a quick*) cover most of what people actually write.

For a long time, bigram and trigram models were good enough to drive practical systems for tasks such as speech recognition, machine translation, and search. They are far from a toy idea. But they have two limitations that no amount of counting can fix, and these limitations are what eventually motivated neural language models.

The first limitation is that the context window is fixed and very small. A bigram model can only "remember" the single previous token. A trigram model can remember two. Increasing the window further makes the count tables explode in size and the data even sparser, so in practice n rarely exceeds 5. This means a bigram model fundamentally cannot use information from earlier in a paragraph, even when that information is exactly what would tell us what the next word should be. If a sentence begins "The student opened her chemistry textbook and started to read about ...", the word *chemistry* is essential for predicting what comes next. Still, by the time the bigram model gets to the blank, all it remembers is the word *about*.

The second limitation is more subtle: bigram models have no notion of word similarity. To the model, *cat* and *dog* are completely unrelated symbols, just like *cat* and *xylophone*. If the training data contains many examples of *the cat sat* but not a single example of *the dog sat*, the model has no way to use the first to inform the second, even though any human reader can see they should behave almost identically. Every word lives in its own isolated context, and generalization across similar words does not happen.

Neural language models were developed to address both limitations at once: they replace fixed windows with learned representations that can carry information from arbitrarily far back, and they map words into a continuous space in which similar words are close together, so that what is learned about one word automatically transfers to related ones.

2.1.3 The Neural Network Solution: Manu

If we cannot memorize the language, we have to learn its structure [BDVJ03]. The chain rule already told us that this reduces to modelling the next token given the previous ones,

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}). \tag{14}$$

For convenience we abbreviate "everything before position t" as

$$P(x_t \mid x_{<t}), \quad \text{where} \quad x_{<t} = (x_1, x_2, \ldots, x_{t-1}). \tag{15}$$

Instead of storing one probability per context, we will compute the probability using a function whose parameters are shared across every context. The cost of the model then grows with the size of the function rather than with the number of possible contexts.

We first need a way to feed text into a function. Since real-valued functions cannot operate directly on words, each token is mapped to a vector of real numbers called a *word embedding* [MCCD13]:

$$x_i \to \mathbf{e}_i \in \mathbb{R}^d, \tag{16}$$

where d is a small fixed number called the embedding dimension, words that appear in similar contexts tend to mean similar things, so the embeddings are designed (and learned) so that similar words land at nearby points in this d-dimensional space [MCCD13].

The context, now represented as a list of vectors, is fed into a neural network f_θ whose behaviour is controlled by a (large) set of parameters θ. The network outputs one real number for every token in the vocabulary,

$$f_\theta(x_{<t}) \in \mathbb{R}^{|V|}. \tag{17}$$

These numbers, called *logits*, are unnormalized scores indicating how plausible each token is to follow. To turn them into a probability distribution we apply the *softmax* function, which exponentiates each score and normalizes the results to sum to one:

$$P_\theta(x_t \mid x_{<t}) = \text{softmax}(f_\theta(x_{<t})). \tag{18}$$

The model now stores only the parameters θ of a single neural network and computes probabilities on demand. The space requirement drops from $O(|V|^T)$, which Section 2.1.2 showed to be astronomical, to $O(|\theta|)$, which is large but tractable on modern hardware. Because the same parameters are reused across every context, the model can also assign sensible probabilities to sequences it has never seen before. Section 2.2 describes what the function f_θ looks like inside, and how its parameters θ are chosen.

2.2 Basics of a Neural Network: Both

- How it mimics a brain: Manu
- Terminology and main components: Manu/Vanshika
- Linear function, then later advanced to non-linear and why (what the gain is): Manu
- Main algorithms: forward and backward propagation: Vanshika

2.2.1 Neural Networks and the Human Brain: Manu

The function f_θ from the previous section is, in practice, almost always a *neural network* (NN) [GBC16]. Neural networks are a flexible family of parameterized functions, and they get their name from a loose analogy with the brain.

The brain is made up of billions of cells called *neurons*, which receive signals from other neurons, combine them, and pass new signals along. The early researchers who built the first artificial networks asked whether a similarly structured system, composed of simple mathematical units rather than biological cells, could also learn from experience. Figure 1 illustrates the analogy between a biological neural network (BNN) and an artificial neural network (ANN).

A modern neural network has only a few moving parts, and the same components appear in every architecture discussed later in this paper.

- **Neurons.** A neuron is the basic computational unit. It takes in several numbers, computes a weighted sum, adds a constant called a bias, and passes the result through a non-linear function called an activation [GBC16].

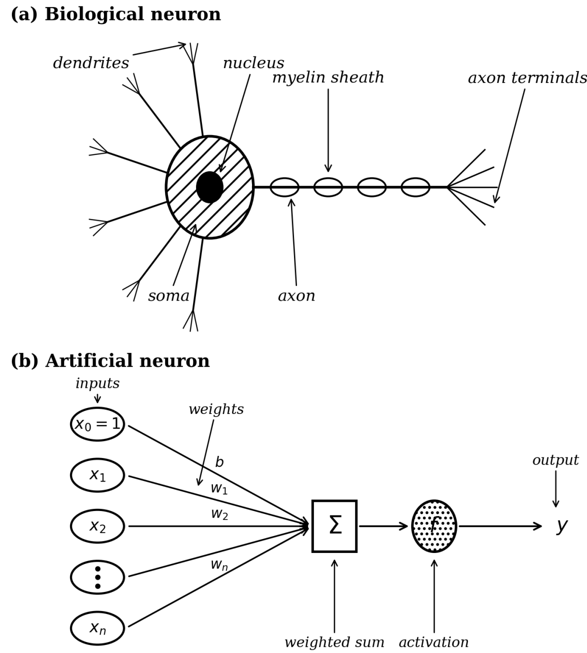

Figure 1: Comparison between a biological neural network (BNN) and an artificial neural network (ANN) [GBC16]

- **Connections.** Connections link neurons, and each connection carries a number called a *weight*. The weight controls how strongly one neuron influences the next: a small weight means little influence, a large weight means a lot.
- **Weights and biases.** The weights and biases are the *parameters* of the network, the numbers that get adjusted during training. Weights set the strength of each connection; biases shift a neuron's output up or down. To see why a bias is useful, consider a function such as $y = \sin(x) + b$: changing b moves the curve up or down without changing its shape, which gives the model an extra degree of freedom for fitting data.
- **Activation functions.** Activation functions are the small nonlinear functions applied at the end of each neuron. They are what allow a network to represent more than straight-line relationships. Common choices are the sigmoid, the hyperbolic tangent, and the rectified linear unit (ReLU). Section 2.2.2 explains why this non-linearity is essential.
- **Learning rule.** A learning rule specifies how the weights and biases are updated when the network makes mistakes. In modern neural networks, this is a gradient-based optimization method, such as stochastic gradient descent [GBC16]; we describe this in detail in Section 2.2.4.

The brain analogy should not be taken too literally. Biological neurons receive electrical and chemical signals from other neurons through structures called *dendrites*, combine those signals inside the cell body, and emit a spike along an outgoing fibre called an *axon* when the combined signal crosses a threshold. Artificial neurons follow the same general recipe of combining inputs and producing an output, but the underlying mechanics are very different. The mechanisms of learning in the human brain are still not fully understood, while artificial neural networks are trained using well-defined mathematical optimization. The biological story is best read as historical motivation rather than a literal claim about either system [GBC16].

2.2.2 Why Non-Linear Activation Functions Are Necessary: Manu

Section 2.2.1 introduced activation functions and noted that they have to be non-linear, but it did not say why. The reason is that without non-linear activations, stacking more layers does

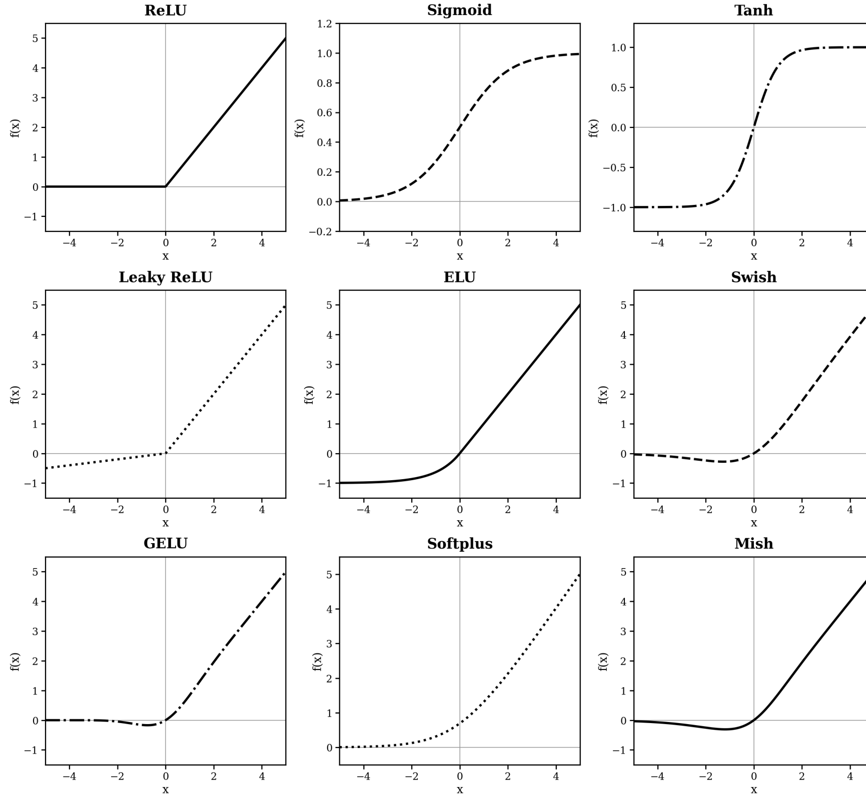

Figure 2: Examples of commonly used activation functions [GBC16]

not make a neural network any more powerful. A deep stack of linear layers is mathematically equivalent to a single linear layer.

The first artificial neural network model, the **Perceptron**, was introduced by Frank Rosenblatt in 1957 [GBC16]. A perceptron computes a weighted sum of its inputs and applies a step function: if the sum is above zero, the output is 1, otherwise it is 0:

$$y = \begin{cases} 1 & \text{if } w^\top x + b > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{19}$$

A perceptron can solve any problem whose answer can be drawn as a single straight line separating two classes, but many real problems are not like this. The standard counter-example is the XOR problem, where no single straight line can correctly separate the inputs $(0, 0)$ and $(1, 1)$ from $(0, 1)$ and $(1, 0)$ [GBC16]. To handle problems like XOR, researchers introduced **Multi-Layer Perceptrons (MLPs)**, which add one or more hidden layers between the input and the output and apply a non-linear activation at each layer. The *Universal Approximation Theorem* provides the theoretical justification for this: a feedforward neural network with at least one hidden layer and a non-linear activation function can approximate any continuous function on a compact subset of $\mathbb{R}^n$, given enough neurons [GBC16].

Why depth without non-linearity adds nothing.

The easiest way to see why non-linearity is necessary is to do the algebra. Suppose we stack two purely linear layers, with no activation function in between:

$$h = W_1 x + b_1, \tag{20}$$
$$y = W_2 h + b_2. \tag{21}$$

Substituting the first equation into the second gives

$$y = W_2(W_1 x + b_1) + b_2, \tag{22}$$

which expands to

$$y = W_2 W_1 x + W_2 b_1 + b_2. \tag{23}$$

Defining a new weight matrix and bias,

$$W' = W_2 W_1, \tag{24}$$
$$b' = W_2 b_1 + b_2, \tag{25}$$

Thee two-layer network reduces to

$$y = W' x + b'. \tag{26}$$

This is just a single linear layer. The same trick works for any number of stacked linear layers: they can always be rewritten as one linear layer with a different weight matrix and bias. A single linear layer can only represent straight-line relationships, so depth on its own provides no additional representational power.

The effect of adding a non-linearity.
Inserting a non-linear activation $\sigma(\cdot)$ between the two layers gives

$$h = \sigma(W_1 x + b_1), \tag{27}$$
$$y = W_2 h + b_2. \tag{28}$$

Because σ is non-linear, there is no algebraic way to merge W_1 and W_2 into a single matrix, and the two layers do separate work. The network can now represent curved, multi-region decision boundaries that no linear model can capture, which is why every modern neural network includes activation functions between its linear layers.

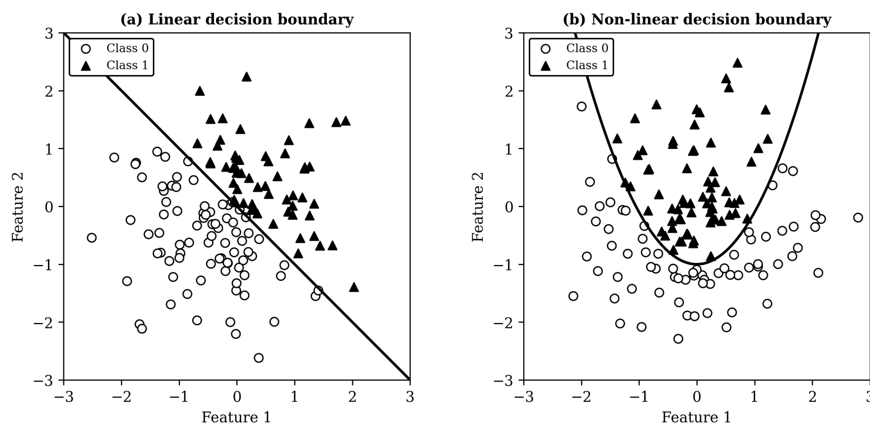

Figure 3: Comparison of linear and non-linear decision boundaries [GBC16]

Feedforward Neural Networks.
Multi-Layer Perceptrons are the simplest example of a more general family called **Feedforward Neural Networks (FFNNs)**. The defining property of an FFNN is that information flows in a single direction, from the input layer through the hidden layers and out the output layer, with no loops or feedback connections.

A forward pass through an FFNN proceeds as follows:

1. Input data is fed into the input layer.
2. Each layer applies a linear transformation followed by an activation function.
3. The result is passed through one or more hidden layers in turn.

4. The final layer produces a prediction.

The structure of the final layer depends on the task:

- **Regression.** For predicting continuous numerical values such as prices or temperatures, the output layer is a single neuron with no activation function, allowing it to produce any real number.
- **Binary classification.** For yes/no problems, the output layer is a single neuron with a sigmoid activation, which maps its input to a probability between 0 and 1.
- **Multi-class classification.** For problems with more than two classes (such as language modelling, where the classes are the vocabulary tokens), the output layer has one neuron per class, and a softmax activation converts the outputs into a probability distribution that sums to 1.

The multi-class case with a softmax output is exactly the setup needed to compute $P_\theta(x_t \mid x_{<t})$ for next-token prediction, as defined in Section 2.1.3.

2.2.3 FFNN intro and architecture - Vanshika

MLP s are FFNN which is already been defined above as a part of linear and non linear section

As we already discussed above how MLPs are the same as feed-forward neural networks (FFNNs), where data flows forward without loops, and they can have one or more hidden layers [GBC16]. Learning in FFNN is basically a process of optimizing the parameter set θ = Weights and Biases be consistent with above terminology and to minimize a cost function $J(\theta)$ [GBC16].

2.2.4 Main Algorithms: Forward and Backward Propagation- Vanshika

In FFNN, learning can be done using two algorithmic passes: the forward pass, which is used to generate a prediction, and the backward pass, which is used to compute the necessary gradients for optimization.[GBC16].

Forward Propagation In the forward pass, input data is passed from the input layer, through one or more hidden layers, to the output layer, Each layer takes the output of the previous one, transforms it, and passes it forward until a final cost is calculated. For a given layer l, the linear transformation and then non-linear activation are computed as shown below:

$$z^{[l]} = W^{[l]} x^{[l-1]} + b^{[l]} \tag{29}$$

$$x^{[l]} = \sigma\big(z^{[l]}\big) \tag{30}$$

where $W^{[l]}$ is the weight matrix, $b^{[l]}$ is the bias vector, and σ is a non-linear activation function (such as ReLU or sigmoid). This process continues until the final output $\hat{y}$ (predicted outcome) is generated, which is thenused to compute the total scalar cost J:

$$J = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \Omega(\theta) \tag{31}$$

In this expression, L is the loss function , which could be a MSE measuring the error between the prediction $\hat{\mathbf{y}}$ and the true target $\mathbf{y}$ from our dataset, while $\Omega(\theta)$ is a regularization term scaled by the hyperparameter λ to prevent overfitting of the parameters θ.[GBC16].

Backward Propagation. After the forward pass, to minimize the loss, the network adjusts its parameters W and b to reduce J [GBC16].

Output Error We begin by finding how the cost changes with respect to the output:

$$\mathbf{g} = \nabla_{\hat{\mathbf{y}}} J \tag{32}$$

This gradient $\mathbf{g}$, is the *error signal* flowing backward from the cost to the inputs. [GBC16].

Back-propagating through Layers For each layer l, from L to 1, this error signal passes through the non-linearity and linear transformation.

- Gradient on Pre-activation not sure if i should add this math, complex - v

$$\mathbf{g} \leftarrow \mathbf{g} \odot \sigma'(\mathbf{z}^{(l)}) \tag{33}$$

- Gradient on Parameters

$$\nabla_{\mathbf{b}^{(l)}} J = \mathbf{g} \tag{34}$$

$$\nabla_{\mathbf{W}^{(l)}} J = \mathbf{g}(\mathbf{a}^{(l-1)})^\top \tag{35}$$

- Gradient on Previous Layer

$$\mathbf{g} \leftarrow (\mathbf{W}^{(l)})^\top \mathbf{g} \tag{36}$$

The symbol $\odot$ denotes the Hadamard product (element-wise multiplication).

These gradients determine how the weights and biases are then updated using an optimization algorithm, such as Stochastic Gradient Descent (SGD),

$$W^{[l]} \leftarrow W^{[l]} - \alpha \frac{\partial L}{\partial W^{[l]}} \tag{37}$$

where α represents the learning rate, a hyperparameter that controls the step size of the update [GBC16].

2.3 How is the Decoder-Only Transformer Architecture different from FFNN : Vanshika

- how is it different from normal transformer?

For our transformer, we will be using decoder only transformers. encoder decoder ... We saw that FFNN in Section 2.2 use fixed layer connections; however, Decoder-Only Transformer evolves this concept by using Self-Attention mechanism[VSP+17], which allows the model to assign different levels of importance to different tokens within an input sequence when generating each output. This enables context awareness understanding in the input Embeddings . For instance, in "The cat sat on the mat because it was soft," the self-attention enables the model to determine that "it" is referring to "the mat."

The Transformer uses masked self-attention to capture interactions between all pairs of tokens in a sequence. This creates direct pathways between every token and every earlier token while still preventing access to future positions, and it allows highly parallel computation during training[VSP+17]. while also maintaining autoregressive flow, where it begins with a start token and uses the list of previous outputs as its inputs, which contain all the attention information from the first input. [VSP+17].

A single decoder block contains the following key components like in the figure:

Masked Multi-Head Self-Attention:

Unlike standard layers in section 2.2, the attention mechanism here allows the future tokens to attend to the prior tokens to gain contextual understanding. Adding Mask M, allows the model to not see or get influenced by future tokens during training [VSP+17]. Formally, attention is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T + M}{\sqrt{d_k}}\right) V \tag{38}$$

where Q, K, V are the Query, Key, and Value matrices, and $\sqrt{d_k}$ is a scaling factor that helps to stabilize gradients.

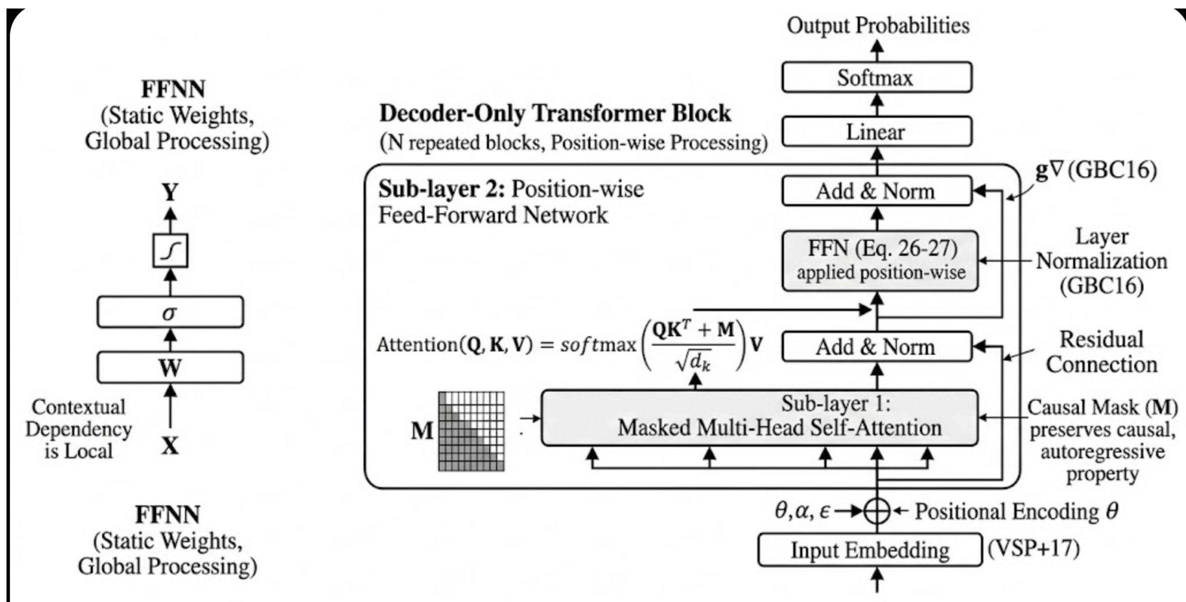

Figure 4: illustrates the internal architecture of a single decoder block, adapted from Vaswani et al. [VSP+17]. TODO: ADD simplified VERSION, THIS IS THE VISION

Position-wise Feed-Forward Network (FFN): After the attention step is performed as labled in sub-layer 2 in the diagram, each token's representation is passed through the same position-wise fully connected network with a non-linear activation function (such as ReLU or GeLU) described in Section 2.2.4 to each position separately and identically[VSP+17]. This helps the model capture more complex patterns than only attention can capture [VSP+17].

$$\mathrm{FFN}(x) = \max(0,\, xW_1 + b_1)W_2 + b_2$$

where x is the single context aware embedding vector coming from the self-attention module. We pass x for each token for better feature extraction and non- linearity.

Layer Normalization and Residual Connections: Each sub-layer (masked self-attention and feed-forward network) is wrapped with a residual connection around it and then a normalization layer[VSP+17]. As labeled in Figure 5 ("Add Norm"), these connections allow the "error signal" **g** (defined in the Backpropagation section 2.2.4) to flow through the architecture without vanishing and allow the total cost J to be optimized efficiently [VSP+17]. These design choices stabilize training, improve gradient flow, and allow us to train much deeper networks [GBC16][VSP+17]

Positional Encoding: Because the transformer does not use recurrence, Self attention module is permutation invariant; it does not know about the token order. Therefore, a positional encoding vector is added to each token embedding at the input stage so that the model can distinguish [VSP+17]

Overall, in comparison to more traditional neural networks, the decoder-only transformer architecture allows highly parallel training, scales effectively with data, and is well-suited for modeling the long-range dependencies when in sequence, Therefore we use the decoder only artitecture for our model[VSP+17].

2.4 Pre-Training versus Post-Training: Manu

2.4.1 Pre-training vs Post-training

The previous sections explained what a language model represents and how a neural network can be used to compute it. They did not explain where the parameters of that network actually come from. In practice, building a modern Large Language Model (LLM) is split into two phases with very different goals: *pre-training* and *post-training*.

Pre-training gives the model its general capabilities: its grasp of grammar, its broad world knowledge, its sense of which words tend to follow which others, and its ability to perform basic reasoning. During pre-training, the model is exposed to large amounts of unlabelled text and learns the statistical patterns of language by repeatedly predicting the next token, exactly as described in Section 2.1.3. No human is labelling each example; the text itself supplies the answer (the next token), which is why pre-training is described as *self-supervised*.

Post-training refines how the model uses what it has learned. After pre-training, a model can produce fluent, coherent text, but it does not necessarily produce *helpful* text, and it does not necessarily refuse to produce harmful text. Post-training addresses this by guiding the model to follow instructions, maintain an appropriate tone, stay on task, and respect safety constraints. It typically uses much smaller, carefully curated datasets and human feedback.

The relationship between the two stages can be thought of as nature and nurture: pre-training gives the model its foundational capabilities, and post-training shapes how those capabilities are expressed in practical interactions.

Key differences.

- **Pre-training:** learning general language patterns and world knowledge from large unlabelled datasets through next-token prediction.
- **Post-training:** adjusting the model's behaviour, instruction-following, tone, and safety using smaller curated datasets and human feedback.

This thesis focuses entirely on pre-training, since our goal is to study how the choice of pre-training corpus shapes the foundational knowledge and biases of the resulting model. We do not perform any post-training. Before moving on to implementation details in Section 3, this subsection walks through what a pre-training pipeline looks like at a high level.

High-level steps in pre-training.

1. **Dataset collection.** A large body of text is gathered from diverse sources such as books, websites, and articles. Since the model can only learn from what it sees, the choice of sources is one of the most consequential decisions in the whole pipeline.
2. **Data processing and tokenization.** The raw text is cleaned, filtered, and converted into tokens by a tokenizer that maps words or subwords to integer IDs, the same tokens defined in Section 2.1.1.
3. **Model initialization.** The neural network architecture (in our case, a decoder-only transformer) is initialized with small random weights.
4. **Self-supervised training.** The model is repeatedly shown chunks of the tokenized text and asked to predict the next token at every position. Because the correct answer is always the next token in the actual text, no manual labelling is required.
5. **Parameter optimization.** A gradient-based optimizer (such as the stochastic gradient descent method introduced in Section 2.2.4) updates the model's parameters to reduce the prediction error, one batch at a time, until the corpus is exhausted.

Section 3 revisits each of these stages and describes how we carry them out for our two philosophical corpora, including the architectural choices, training infrastructure, and the procedure for comparing Eastern and Western training data.

3 Method

3.1 Pre-Training Steps: Vanshika

3.1.1 Architecture and Tokenization

We selected **GPT-2** as our baseline model because it is open source and follows the standard decoder-only transformer design with traditional Multi-Head Attention and core autoregressive properties [RWC+19]. This architecture is also well understood and widely studied, which lets us evaluate our results when training from scratch on philosophical text. We chose GPT-2 with three distinctions that improve deep-layer convergence:

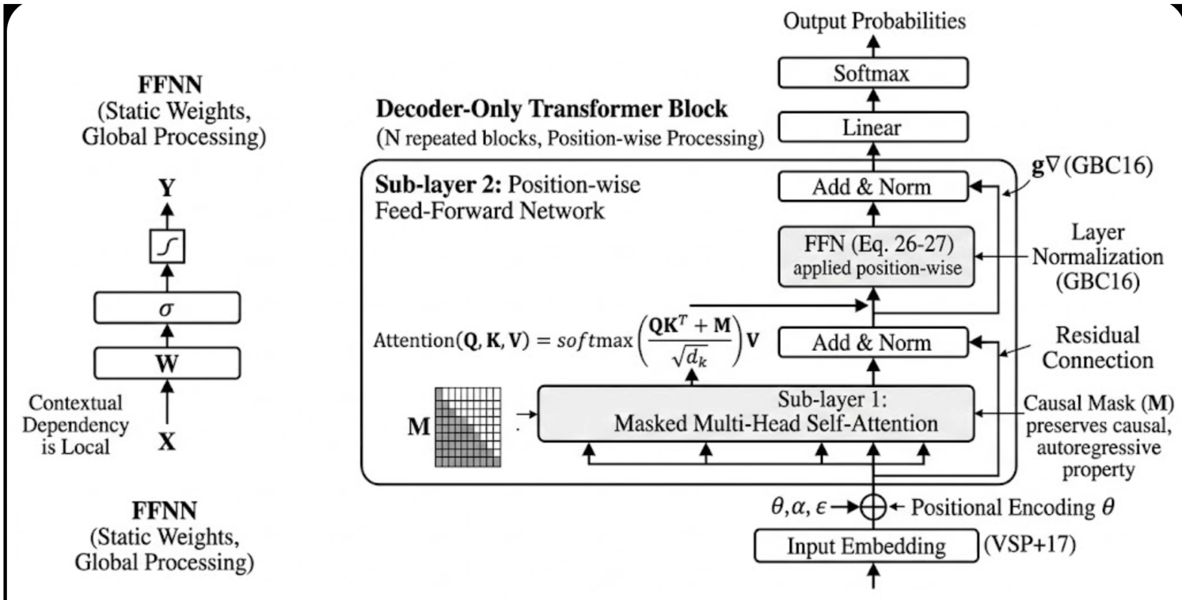

Figure 5: Comparison between a vanilla decoder and the modified version used by GPT-2, from Vaswani et al. [VSP+17] and [RWC+19]. TODO: add simplified version after checking with prof.

- **Pre-Layer Normalization:** Unlike the standard decoder, which applies normalization after the residual addition (Post-LN) [VSP+17], GPT-2 moves Layer Normalization to the input of each sub-block (Pre-LN) [RWC+19]. This creates a cleaner identity path for gradients, which is essential when training the model's deeper variants.
- **Activation Function:** As shown in Figure 3, GPT-2 uses Gaussian Error Linear Units (GELU) [RWC+19] instead of the ReLU activation used by vanilla decoder-only transformers [VSP+17]. GELU provides a smoother, non-monotonic curve that has been shown to improve performance in high-complexity language tasks [HG16].

  Original FFN: $\text{ReLU}(xW_1 + b_1)W_2 + b_2$

  GPT-2 FFN: $\text{GELU}(xW_1 + b_1)W_2 + b_2$
- **Modified Residual Initialization:** The weights of the residual layers are scaled initially by a factor of $1/\sqrt{N}$, where N is the total number of layers [RWC+19]. This accounts for the natural buildup of variance in the residual connections as they move through the stack. By making each residual layer contribute a little less at the start, the model keeps the variance stable across the network, which makes the gradient move more smoothly during backpropagation [RWC+19, XYH+20].

GPT-2 Scaffold: we use the `tiktoken` library for efficient Byte-Pair Encoding (BPE). We initialize with random weights to ensure that the results are derived exclusively from our training set.

3.1.2 Data Formatting and Pipeline

To ensure consistent ingestion, the training corpus is pre-processed to UTF-8 plain text and stored in a centralized `data/` directory.

- **Sharding Strategy:** the corpus is split into sequential shards (e.g. `shard-0001.txt`, `shard-0002.txt`).
- **Concatenation:** during initialization, the training script (`gpt2_pretrain.py`) ingests these files in sorted alphanumeric order. The text is concatenated into a single continuous stream, allowing the model to learn dependencies that span individual file boundaries.

3.1.3 Initialization and Training Loop

Unlike fine-tuning approaches, the model is initialized with random weights (a "from scratch" model) so that all learned linguistic and philosophical representations are derived solely from our Western and Eastern corpora.

- **Trainer Implementation:** we use a customized `Trainer` class to manage the optimization process.
- **Execution:** the training loop is triggered via `trainer.train()`, which processes the concatenated data stream in batches.
- **Artifact Storage:** model checkpoints, tokenizer configurations, and training logs are serialized to the `outputs/` directory for subsequent evaluation.

3.2 Data Collection: Manu

To evaluate whether models trained on distinct philosophical traditions develop measurably different linguistic and conceptual biases, we curated two large-scale corpora: a **Western** corpus spanning Greco-Roman, Christian, and modern European philosophy, and an **Eastern** corpus covering Chinese, Indian, Japanese, Korean, and Tibetan philosophical traditions. In total the source catalogue contains 378 entries, of which approximately 181 were successfully downloaded as public-domain English text. The remaining 197 are documented with reasons for unavailability (e.g. no public-domain English translation, copyright restrictions, or lost primary works).

3.2.1 Sourcing Strategy: Manu

All texts were collected using an automated download pipeline (`download_pretraining_data.py`) that draws from five public-domain sources:

- **Project Gutenberg:** used for high-quality, boilerplate-stripped plain-text editions of classical works (e.g. Griffith's *Rigveda*, Jowett's Plato, Legge's Confucius). Both direct downloads by book ID and keyword-based search via the Gutendex API were employed.
- **Internet Archive:** used for academic and historical translations not available on Gutenberg, retrieved via the Advanced Search JSON API with automatic selection of the best available `.txt` file per item.
- **SuttaCentral:** used for Bhikkhu Sujato's modern English translations of the Pali Canon (Dīgha, Majjhima, Saṃyutta, Aṅguttara, Khuddaka Nikāyas, and Dhammapada), accessed through the bilara-data API.
- **Access to Insight:** used for additional Theravāda Buddhist suttas, collected via recursive HTML scraping of the Tipiṭaka index pages.
- **Sacred-Texts.com:** used selectively for texts not available through the above sources, collected via subpage traversal and HTML-to-text conversion.

Every downloaded text passes through a post-processing pipeline that (i) strips Project Gutenberg boilerplate headers and footers, (ii) converts HTML to plain text where necessary, (iii) normalizes whitespace and line breaks, and (iv) applies an English-language filter based on ASCII ratio, Latin-alphabet ratio, and stopword frequency to reject non-English or heavily corrupted downloads.

3.2.2 Temporal Alignment and Corpus Composition: Manu

A critical challenge in comparative philosophy is separating cultural differences from temporal ones. Differences often attributed to "East vs. West" may actually be driven by historical era or translation style. To address this, both corpora are organized into 21 chronological time periods: one pre-Common Era folder (`older (BC)`) and twenty 100-year intervals from 100 CE through 2000 CE. Each text is assigned to a period based on the earliest known composition date of the original work, not the date of translation. This structure enables the progressive training procedure described in Section 3.5 and lets us control for temporal confounds when comparing Eastern and Western outputs.

The Western Corpus spans approximately 142 philosophers and covers the full arc of European and Mediterranean thought. Representative texts include:

- **Classical Antiquity:** Pre-Socratic fragments, Plato's dialogues (Jowett translations), Aristotle's major works, Epicurus, Lucretius's *De Rerum Natura*, Cicero, Seneca, Epictetus's *Discourses*, and Marcus Aurelius's *Meditations*.
- **Late Antiquity and Medieval:** Plotinus's *Enneads*, Augustine's *Confessions* and *City of God*, Boethius's *Consolation of Philosophy*, Anselm's *Proslogion*, Aquinas's *Summa Theologiae*, as well as Islamic-tradition philosophers classified under the Western canon (Al-Kindi, Avicenna, Al-Ghazali).
- **Early Modern to Modern:** Descartes, Spinoza, Locke, Hume, Kant, Hegel, Kierkegaard, Nietzsche, William James, Bertrand Russell, Wittgenstein, Heidegger, Sartre, Camus, and Simone de Beauvoir, among others.

Also included is the King James Version of the Bible (1611) as a foundational Western religious text.

The Eastern Corpus spans approximately 215 entries across five geographic traditions:

- **Chinese Philosophy** (≈96 entries): Confucian classics (Analects, Mencius, Xunzi), Daoist texts (Tao Te Ching, Zhuangzi, Liezi), Legalist works (Han Feizi, Shang Yang), Mohist texts, Neo-Confucian writings (Zhu Xi, Wang Yangming), and Chan/Zen texts (Platform Sutra, Huineng).
- **Indian Philosophy** (≈78 entries): Vedic texts (Rigveda, Upanishads), the Pali Canon (six collections via SuttaCentral), the Bhagavad Gita, Jain sutras (Jacobi's SBE translations), Nyāya, Sāṅkhya, and Yoga sūtras, as well as Mahāyāna Buddhist works (Nāgārjuna, Vasubandhu, Asaṅga).
- **Japanese Philosophy** (≈16 entries): Zen Buddhist texts (Dōgen, Hakuin), Shinto classics (Kojiki, Nihon Shoki), and Bushidō literature.
- **Korean Philosophy** (≈10 entries): Neo-Confucian works (Yi Hwang, Yi I) and Korean Buddhist texts (Wonhyo, Jinul).
- **Tibetan Philosophy** (≈14 entries): texts from Milarepa, Tsongkhapa, Gampopa, and Longchenpa.

3.2.3 Handling Unavailable Texts: Manu

Of the 378 catalogued entries, 197 could not be downloaded because no public-domain English translation exists, the only available translations remain under copyright, the primary work survives only as fragments within other texts already in the corpus, or access was restricted by the hosting platform. Each unavailable entry is documented in the source catalogue with a machine-readable reason code (e.g. `no_pd_english`, `copyright_trans`, `lost_work`, `restricted_ia`), ensuring full reproducibility and transparency about corpus coverage gaps.

3.3 Environment Setup: Manu

All training and evaluation runs were carried out on **cmps03**, a shared GPU server hosted by the UBC Okanagan Department of Computer Science. The machine is equipped with two **NVIDIA RTX 6000 Ada Generation** GPUs (48 GB VRAM each), running Ubuntu Linux with CUDA 12.x. Training jobs were launched on a single GPU at a time to keep memory headroom for the 124M-parameter GPT-2

architecture at a 1024-token context length, with the second GPU reserved for concurrent evaluation runs.

The software stack consists of Python 3.10 with PyTorch, the Hugging Face `transformers` and `datasets` libraries, the `tiktoken` tokenizer, and a small in-house `lm_utils` module that wraps dataset construction, blockification, and the `Trainer` loop. All artifacts (model checkpoints, tokenizer configurations, training logs, and evaluation JSONs) are written under `~/outputs_full/` on the server, organized by region and time period (e.g. `progressive_west/period_2000/checkpoint-2118`).

The full progressive training pipeline (21 sequential time periods, separately for the Western and Eastern corpora) takes on the order of 24–36 hours of wall-clock time per region on a single RTX 6000 Ada, with most of that time spent on the later, larger time periods. Evaluation runs across all 14 prompt categories and 284 prompts, completing in roughly 30–45 minutes per checkpoint.

3.4 Evaluation Metrics: Manu

Because both models are trained from random initialization on disjoint corpora, standard intrinsic metrics such as same-model perplexity are not directly comparable across the two systems. We instead evaluate the trained models along three complementary axes: *linguistic divergence*, *lexical/stylistic quality*, and *conceptual bias*. All metrics are computed on text generated from a fixed set of 284 philosophical prompts grouped into 14 thematic categories (e.g. *self_identity*, *ethics_morality*, *reality_existence*, *death_immortality*, *enlightenment_liberation*, *free_will_fate*). Generation uses a fixed decoding configuration (`temperature`=0.4, `top_p`=0.9, `repetition_penalty`=1.3, `no_repeat_ngram_size`=4) tuned to suppress degenerate looping in the small from-scratch models.

3.4.1 Cross-Perplexity (Primary Metric): Manu

Our primary metric is **cross-perplexity**: the perplexity assigned by one model to text generated by the other. Specifically, we compute $\mathrm{PPL}_{\mathrm{west}}$(east-output) and $\mathrm{PPL}_{\mathrm{east}}$(west-output). If the two models have learned distinct linguistic and conceptual distributions, each should find the other's output highly improbable, yielding large cross-perplexity values. If they have converged on similar distributions, cross-perplexity will be low. This formulation lets us quantify model divergence directly from output behaviour without relying on a shared held-out test set.

3.4.2 Lexical and Stylistic Metrics: Manu

To complement cross-perplexity, we report:

- **Type–Token Ratio (TTR):** the ratio of unique tokens to total tokens in generated outputs, used as a coarse measure of lexical diversity. Low TTR values indicate that a model is collapsing into a small repeated vocabulary, a known failure mode of overfit small language models.
- **Repetition rate:** the fraction of repeated n-grams in generated text, used together with TTR to detect degenerate outputs that would otherwise inflate perceived novelty.

These two metrics were essential during model selection: the 5,000- and 15,000-step checkpoints achieved low cross-perplexity but also collapsed in TTR and repetition, indicating overfitting rather than learning. The 2,000-step checkpoint was chosen as the strongest result because it preserved lexical diversity while still showing clear cross-distributional divergence.

3.4.3 Concept Marker Frequency: Manu

To test whether each model has internalized the conceptual vocabulary of its training tradition, we maintain two curated marker lists:

- A **Western marker list** containing terms such as *soul, virtue, reason, logos, justice, substance, categorical, dialectic, Kant, Hegel, Aristotle*, etc.
- An **Eastern marker list** containing terms such as *dharma, karma, nirvana, atman, brahman, tao, qi, wu wei, sunyata, dukkha, anatta, mindfulness*, etc.

For each generated response, we compute the relative frequency of Western vs. Eastern markers and aggregate per category. The expected pattern is that the Western model produces a significantly higher Western-marker ratio than the Eastern model, and vice versa. Asymmetries in this ratio are reported per prompt category to surface where each tradition dominates the generated output.

3.4.4 Distributional Overlap: Manu

Finally, to characterize the global similarity of the two models' output distributions, we compute the **Bhattacharyya coefficient** and **Bhattacharyya distance** between the empirical word distributions of Western and Eastern generations, along with a **KL-divergence** report (`kl_divergence.py`) over high-frequency vocabulary. These distributional measures provide a single corpus-level summary that complements the per-category breakdown above and is interpreted on a coarse five-level scale from "very high overlap" to "very low overlap."

All evaluation outputs are serialized to timestamped JSON files (e.g. `bias_evaluation_20260410_062642.json`) under `progressive_evaluations/` for downstream analysis and figure generation.

3.5 Progressive Training Pipeline: Manu

The actual training is driven by `main_progressive_training.py`, which trains the Western and Eastern GPT-2 models in lockstep across the 21 chronological time periods defined in Section 3.2. This subsection describes how that pipeline works and why each piece is there.

3.5.1 Why Progressive, Not Single-Shot

We could have trained one Western model on all Western texts at once and one Eastern model on all Eastern texts at once, in a single training job per region. The progressive pipeline does it differently: it trains 21 separate models per region, where the model for period p is trained on all text from period p and earlier. The model for *older (BC)* sees only pre-Common Era texts, the model for period *100* sees BC plus 0–100 CE texts, and so on up through period *2000*, which sees everything.

There are two reasons to do it this way. Each period yields its own checkpoint and evaluation, so we get a series of snapshots of how the bias signal develops over time, not just a single endpoint. And small early corpora and large late corpora go through the same code path, which forces the training-budget logic to scale with corpus size instead of being hard-coded for one particular size.

3.5.2 Per-Period Training Loop

For each time period, and for each region (east and west), the pipeline does the following:

1. **Load cumulative texts.** `load_cumulative_texts` walks every period directory from `older (BC)` up to the current period and concatenates all the texts under `data/{region}/{period}/`. The data layout from Section 3.2 is what makes this a one-line operation.

2. **Build a fresh model from scratch.** A new GPT-2 (12 layers, 12 heads, 768 embedding dimensions, 1024 block size) is constructed using `build_gpt2_from_scratch`, with random weights and a fresh tiktoken-based tokenizer. We do not warm-start from the previous period's checkpoint, because doing so would conflate "the model learned this from earlier texts" with "the optimizer state was already partway to a solution."

3. **Compute the step budget.** The number of training steps is set dynamically by `compute_max_steps`, which scales linearly with corpus size at 20 steps per text and clamps the result to the range $[300, 5000]$. The early periods, which have only a handful of texts, get a few hundred steps; the late periods, which have hundreds of texts, hit the 5,000-step cap.

4. **Apply the epoch cap.** After the dataset has been built, `compute_epoch_capped_steps` computes how many steps would constitute three full passes over the dataset and reduces `max_steps` to that ceiling if necessary. This is the cap discussed in Section 5: it was added after early small-corpus runs at 5,000 and 15,000 steps memorized their training data verbatim instead of learning a distribution.

5. **Train via Hugging Face `Trainer`.** `build_trainer` (in `lm_utils.py`) wraps the model, dataset, and collator in a standard `Trainer` configured with the hyperparameters from `DEFAULT_CONFIG` (learning rate 5×10^{-4}, per-device batch size 8, gradient accumulation 4, effective batch size 32). Checkpoints are written every ∼10% of the run.

6. **Resolve the final checkpoint.** After training completes, the latest `checkpoint-N` folder under `outputs/progressive_{region}/period_{period}/` is selected as the period's official checkpoint and recorded in the training manifest.

3.5.3 Per-Period Evaluation

Once both the Western and Eastern checkpoints are available for a given period, the pipeline automatically runs the evaluation suite from Section 3.4 against them. `evaluate_bias.py` is invoked with both checkpoints and writes a timestamped JSON file containing per-prompt cross-perplexities, marker counts, TTRs, repetition scores, and KL divergences for all 284 prompts. `batch_chat_test.py` is then invoked once per region to dump raw generations for inspection. By the end of a full pipeline run, each of the 21 time periods has both a pair of checkpoints and a complete evaluation, all under `outputs/progressive_evaluations/period_{period}_evaluation/`.

3.5.4 Bookkeeping and Resume Support

The pipeline writes a top-level `training_manifest.json` that records the timestamp, the resolved Eastern and Western checkpoint paths, and the full training config used for each completed period. This serves two purposes. It is the source of truth for which checkpoint corresponds to which period (later analysis scripts read it instead of guessing from directory names), and it makes the pipeline restartable: if a run is interrupted, `--resume-from <period>` skips ahead to the named period and continues from there without redoing earlier work. Combined with the cumulative-loading design, this means a long pipeline run can be paused, resumed across days, or rerun for a single period in isolation, without the rest of the pipeline knowing or caring.

3.5.5 What the Pipeline Does Not Do

A few things are out of scope. The pipeline does not do hyperparameter search: the values in `DEFAULT_CONFIG` are fixed across all periods and both regions. It does not do early stopping on validation loss; training runs until `max_steps` is reached. There is no post-training, fine-tuning, or RLHF, in line with the scope set out in Section 2.4. Comparing the resulting checkpoints with those from a different training procedure is handled separately in Sections 4.2 and 4.3.

4 Results

We evaluated the progressive pipeline's *period_2000* checkpoints across all 284 prompts and 14 thematic categories.

4.1 Progressive Pipeline Checkpoints

The progressive pipeline trains 21 cumulative models per region, one per chronological time period. The model for period p is trained on all text from period p and earlier, so the *period_2000* model has seen everything from pre-Common Era texts through the 20th century, processed as a structured sequence of incremental passes rather than a single flat stream.

For each time period, the pipeline loads all cumulative texts up to that period, initialises a fresh GPT-2 from random weights, and sets a dynamic step budget that scales linearly with corpus size (20 steps per text, clamped to the range [300, 5,000]). An epoch cap limits training to three full passes over the data. Each period produces its own checkpoint, giving 21 temporal snapshots of how distributional bias develops as more text is added. The checkpoints we report here are the final ones:

- **Western model:** `progressive_west/period_2000/checkpoint-2118`
- **Eastern model:** `progressive_east/period_2000/checkpoint-1299`

The step counts (2118 and 1299) are lower than what a flat step budget would produce. The epoch cap prevented any additional passes once each period's data had been covered three times, which was intentional.

How the epoch cap was introduced. The initial pipeline version used only the dynamic step budget with no hard epoch ceiling. On early small-corpus periods this caused memorisation: with only a handful of texts available in the earliest time periods, even a modest step count translated to over 120 effective epochs through the data. Loss dropped below 0.5 and inference produced near-verbatim regurgitation of training text. Once we checked the effective epoch count and saw how far out of range it was, we added `compute_epoch_capped_steps` as a hard ceiling so training would stop after three full passes regardless of what the step budget computed. That change fixed the early periods and is why the final checkpoint step counts (2118 and 1299) look low relative to the step budget.

4.1.1 Output Quality

These checkpoints are the first to avoid degenerate output, show genuine distributional separation, and produce tradition-aligned vocabulary at the same time. Mean TTR is **0.821** (Western) and **0.850** (Eastern) across 284 responses, with a repetition score of **0.000** for both. Cross-perplexity is large in both directions (Section 4.1.2), and concept-marker ratios lean toward each model's own tradition (Section 4.1.3).

4.1.2 Cross-Perplexity

Across all 284 prompts and 14 categories, the Western model assigns a mean perplexity of **13,722** (median 11,658) to Eastern-generated text, while the Eastern model assigns a mean perplexity of **2,288** (median 1,901) to Western-generated text. Both are well above what either model assigns to its own output.

The $\approx 6\times$ gap between the two directions comes down to translation register. Most of the Eastern corpus arrives as 19th-century or modern English translations (Legge, Griffith, Sujato), and that translation English overlaps substantially with the modern English in the Western corpus. So when the Eastern model reads Western-generated text, much of the vocabulary is already familiar. The Western corpus is weighted toward narrower registers: the KJV, Jowett's Plato, Augustine, Kant. Text from those registers looks genuinely foreign to the Eastern model.

4.1.3 Concept Marker Bias

Averaged over all generated responses:

- In the **Western model's** output, Western markers (*soul, virtue, reason, logos, justice, substance, dialectic*, etc.) account for **77.7%** of all marker hits; Eastern markers account for **11.1%**.
- In the **Eastern model's** output, Eastern markers (*dharma, karma, nirvana, atman, tao, sunyata, mindfulness*, etc.) account for **52.7%** of marker hits; Western markers account for **37.4%**.

Both models lean toward their own tradition, but the Western model does so more strongly. This goes back to the same translation register issue: translators of Eastern texts regularly use Western philosophical terms as glosses, so the Eastern model was exposed to both vocabularies during training and uses them in roughly the proportions they appeared in the corpus.

Per-Category Breakdown. Table 1 breaks the marker ratios down across all 14 thematic categories.

The largest gaps show up in categories where Eastern traditions built technical vocabulary with no real Western equivalent: *enlightenment_liberation* (E-mod E% = 0.68 vs. W-mod E% = 0.07), *wisdom_truth* (0.72 vs. 0.24), *death_immortality* (0.64 vs. 0.11), *body_mind* (0.66 vs. 0.06), and *good_evil* (0.62 vs. 0.05). Terms like *nirvana*, *moksha*, *sunyata*, *anatta*, and *dukkha* show up regularly in the Eastern model's outputs and almost never in the Western model's. The smaller gaps in *free_will_fate* and *self_identity* make sense given that both traditions discuss those ideas with heavily overlapping vocabulary.

Category	W-mod W%	W-mod E%	E-mod W%	E-mod E%
self_identity	0.79	0.11	0.61	0.39
purpose_meaning	0.75	0.15	0.38	0.52
ethics_morality	0.90	0.05	0.33	0.52
reality_existence	0.76	0.14	0.56	0.29
knowledge_truth	0.75	0.15	0.33	0.53
death_immortality	0.80	0.11	0.27	0.64
nature_universe	0.86	0.10	0.50	0.42
enlightenment_liberation	0.88	0.07	0.27	0.68
free_will_fate	0.68	0.17	0.47	0.38
good_evil	0.75	0.05	0.23	0.62
society_justice	0.87	0.08	0.38	0.47
death_meaning	0.79	0.06	0.28	0.57
body_mind	0.59	0.06	0.34	0.66
wisdom_truth	0.71	0.24	0.27	0.72

Table 1: Mean concept-marker ratios per category for the progressive pipeline checkpoints. "W-mod W%" = fraction of Western markers in the Western model's output; "E-mod E%" = fraction of Eastern markers in the Eastern model's output.

4.1.4 Vocabulary and Distributional Overlap

Figure **??** shows the vocabulary bubbles for the progressive checkpoints. Eastern-specific terms (*buddha*, *reborn*, *insight*, *mind*, *suffering*) appear as large red bubbles with no counterpart in the Western panel. The Western panel is dominated by dialogue markers (*say*, *said*, *know*) alongside tradition-specific terms (*soul*, *true*, *nature*). The combined panel shows a purple core of shared high-frequency prose with coloured peripheries that barely touch.

4.1.5 Distributional Overlap and the Long-Tail Effect

Computed over all generated tokens, the two progressive models have a **Bhattacharyya coefficient of 0.8298** (Bhattacharyya distance 0.1866). Every individual category falls in the same high-overlap range (BC 0.72–0.81), and the mean symmetric KL divergence on prompt-conditioned distributions is only **9.29%**.

This might seem to contradict the marker results, but the two metrics are measuring different things. The Bhattacharyya coefficient is dominated by the most frequent tokens, which in any fluent English text are function words (*the*, *and*, *is*, *that*). Both models produce those at similar rates regardless of philosophical content, so the BC stays high. The tradition-specific vocabulary sits in the low-frequency tail where the BC computation barely picks it up.

The raw vocabulary counts make this concrete. Across all generated text, the Western model uses **1,049** unique words and the Eastern model uses **1,148**, with only **489** shared. Around 60% of each model's vocabulary is not in the other's output at all, but because those private words are low-frequency, the shared 40% dominates the BC. A very low BC would mean the models had drifted into incompatible dialects, which would suggest overfitting. What we see instead is a shared high-frequency base of ordinary English with tradition-specific vocabulary sitting in the tail, which is exactly where the marker lists are looking.

5 Conclusion

Training identical GPT-2 architectures on philosophically distinct corpora does produce measurably different distributional biases, and the progressive pipeline's *period_2000* checkpoints show this most clearly.

The main finding is a consistent cross-perplexity asymmetry. The Western model assigns roughly six times higher perplexity to Eastern-generated text than the Eastern model assigns to Western-generated text (13,722 vs. 2,288 mean perplexity). This comes from the corpus composition rather than any quirk of the evaluation setup. Eastern philosophical texts arrive primarily through 19th- and 20th-century English translations that share a lot of vocabulary with the Western corpus, so the

Eastern model was exposed to both registers during training. The Western corpus is weighted toward narrower, older registers (the KJV, Jowett, Kant) that the Eastern model never encountered. The resulting difference in distributional familiarity is detectable through cross-perplexity.

Concept-marker bias holds across all 14 thematic categories. The Western model puts 77.7% of its marker usage toward Western terms; the Eastern model puts 52.7% toward Eastern ones. The weaker Eastern lean is consistent with the translation register explanation: terms like *dharma* and *nirvana* appear in both corpora, so the Eastern model learned them alongside Western philosophical vocabulary and uses both. The Western model had no comparable exposure to Eastern terminology and rarely produces it.

The epoch cap ended up being the most consequential decision in the project, even though it was not part of the original design. Without a ceiling on training passes, small-corpus periods memorised their data rather than learning a distribution, and the resulting models could not be meaningfully compared. Capping at three passes gave the models enough exposure to absorb the corpus register without pushing into regurgitation. Projects doing corpus-conditioned language model training should probably treat epoch count as a first-class hyperparameter rather than an afterthought.

6 Future Work

A few directions stand out as worth pursuing.

Larger base models. GPT-2 at 124M parameters works for a controlled corpus experiment, but its vocabulary and context window are small enough that some philosophical nuance in the training texts may not survive pre-training. Running the same progressive pipeline with GPT-2 Medium or Large would test whether the asymmetry patterns scale up or whether they are partly an artefact of the small model's capacity limits.

Cleaner corpus boundaries. The cross-perplexity asymmetry is partly a translation artefact since Eastern texts arrive in English register that overlaps with Western texts. A cleaner test would retrain the Eastern model on texts originally written in English (modern comparative philosophy, contemporary scholarship) separately from texts in original languages (Pāli, Sanskrit, Classical Chinese), then check whether the asymmetry changes when translation vocabulary is taken out.

Temporal bias trajectories. This paper only reports the final *period_2000* checkpoint, but the pipeline produces 21 checkpoints per tradition. Plotting cross-perplexity and marker bias across all 21 periods would show when the two models actually diverge: whether the split happens early with pre-Common Era texts, builds gradually, or is driven by a few high-influence periods.

Probing and interpretability. The marker analysis works at the surface level. A more informative test would use linear probes on the model's hidden states to check whether tradition identity is encoded in the internal representations directly, and if so, which layers carry the signal.

Multilingual extension. Running the Eastern pipeline on original-language texts with a multilingual tokeniser and evaluating in those languages would remove the translation confound entirely. It would also clarify whether the cross-perplexity asymmetry reflects something about the philosophical traditions themselves or is mainly a consequence of how they were rendered into English.

References

[BDVJ03] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 2003.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[Gol16] Yoav Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 2016.

[HG16] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[JM23] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. 2023. Draft available online.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[RWC+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Technical Report*, 2019.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.

[XYH+20] Ruibin Xiong, Yunchang Yang, Di He, Kaiheng Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture. In *International Conference on Machine Learning (ICML)*, pages 10524–10533. PMLR, 2020.