Introduction to LLM for predictions in Economics: An example with LABOR-LLM (Athey et al., 2024)

Federico Nutarelli

[1] IMT School for Advanced Studies Lucca, Visiting TSE, MIT Sloan

Introduction

- **Who am I?** (i) Enthusiast of causal Machine-Learning and Empirical IO, (ii) a slides reader;

Introduction

- **Who am I?** (i) Enthusiast of causal Machine-Learning and Empirical IO, (ii) a slides reader;
- **Who am I NOT?** (i) a Labor economist; (ii) a synthetic person in slides (iii) a coherent persons in punctuation and aesthetic of slides (iv) someone that stick with timings in lectures (work in progress, do not copy my style in presentations);

Introduction

- **Who am I?** (i) Enthusiast of causal Machine-Learning and Empirical IO, (ii) a slides reader;
- **Who am I NOT?** (i) a Labor economist; (ii) a synthetic person in slides (iii) a coherent persons in punctuation and aesthetic of slides (iv) someone that stick with timings in lectures (work in progress, do not copy my style in presentations);
- **Why am I here?** To hopefully provide you with a tool for Labor related research questions;

Introduction

- **Who am I?** (i) Enthusiast of causal Machine-Learning and Empirical IO, (ii) a slides reader;
- **Who am I NOT?** (i) a Labor economist; (ii) a synthetic person in slides (iii) a coherent persons in punctuation and aesthetic of slides (iv) someone that stick with timings in lectures (work in progress, do not copy my style in presentations);
- **Why am I here?** To hopefully provide you with a tool for Labor related research questions;
- **What are these slides about?** A very simple introduction to LLMs + the way to use transformers (GPT architecture) to **predict** next job. Athey et al. (2024) use old GPTs but similar structure of modern (but no thinking mechanism);

Introduction

- **Who am I?** (i) Enthusiast of causal Machine-Learning and Empirical IO, (ii) a slides reader;
- **Who am I NOT?** (i) a Labor economist; (ii) a synthetic person in slides (iii) a coherent persons in punctuation and aesthetic of slides (iv) someone that stick with timings in lectures (work in progress, do not copy my style in presentations);
- **Why am I here?** To hopefully provide you with a tool for Labor related research questions;
- **What are these slides about?** A very simple introduction to LLMs + the way to use transformers (GPT architecture) to **predict** next job. Athey et al. (2024) use old GPTs but similar structure of modern (but no thinking mechanism);
- **What are these slides NOT about?** The very formal mathematical structure of transformers (just intro).

LLM in Economics

Ludwig, Mullainathan, Rambachan (2025) (LMR) distinguish 2 types of econometrically justified tasks of LLMs in Economics:

Prediction tasks: valid under one no "leakage" condition between the LLM's training dataset and the researcher's sample. No leakage can be ensured by using open-source LLMs with documented training data and published weights. $\rightarrow$ usually problems are bias and representativeness (e.g. are LLM responses representative of the population ones?)

Estimation problems + counterfactuals: more complex. If interested next year.

What about today's lecture?

Introduction to the world of LLMs with an example on Labor Economics (Athey et al., 2024). Athey et al. (2024) uses LLMs as a predictive task.

Claim of the paper: "by using LLMs rather than a fine-tuned transformer architecture as in CAREER, we can obtain better job-transition probabilities."

The first step is to understand what is a transformer and what is an LLM...

LLMs in a snapshot

At their most basic LLMs are statistical pattern-recognition and prediction systems

LLMs output the next likely word (“token”) in a sentence (“sequence”).

(i) token: unit of text e.g. word, character. 1 word 0.75 token
(ii) sequence: context or section (“window”) of text e.g. sentence, paragraph, book
(iii) Max seq. length into GPT2 is 4096 tokens; Claude 2 is 100K tokens

The likelihood of the next word appearing is determined by the context in which the words are seen in a larger body of text (“corpus”) and the input to the chat (user given)

Learning from a large corpus (see TRAINING LLM) allows LLMs to understand the meaning of words.

LLMs derive patterns (meaning) from extensive amounts of data (training on vast and varied corpora).

For instance, consider the large number of sentences an LLM might encounter that begin with the phrase "my favourite colour is...". Given this training, an LLM can predict with a high degree of certainty that the next word is likely to be a colour (although it doesn't know what "colour" means the way we do).

This continual exposure enables LLMs to cluster or group words like “red, blue, green. . . ” into a collective set that represents the abstract concept of “colour”.

However "understanding" of LLMs is fundamentally different from human comprehension: LLMs generate statistical patterns, grouping similar tokens based on complicated metrics that determine similarity or dissimilarity between tokens. It’s less about true comprehension and more about recognizing patterns and connections from vast amounts of data.

TRAINING LLM

- LLMs are trained in an unsupervised manner (unlabeled data) on vast quantities of open source and licensed data e.g. The Pile (825GB, incl. web, papers, patents, books, ArXiv, Stack Exchange, maths problems, computer code)
- Example nr. parameters of transformer: GPT3: 175B parameters;
- Responses (and parameters) are refined using question-response pairs ("InstructGPT") from the web, humans or bootstrapped (i.e. the LLM outputs its own pairs) $\rightarrow$ reinforcement learning with human feedback (RLHF) is used to reward LLMs to give appropriate responses ("guardrails" like privacy, human right, ethics,...)

Prediction

- **Token Prediction**: is influenced by the frequency the word is seen in various contexts but there is a degree of randomness so that the word with the highest probability isn't always seen.
- For each token, the model creates a **probability distribution** over all possible tokens in its vocabulary.
- **Example:** If given the prompt "The sky is . . . ", the model might assign probabilities like:

Token	Probability
blue	60%
clear	20%
cloudy	10%
gray	5%
other	5%

Table 1: Example of (decoder) output.

- The next token is selected based on this distribution, which can vary according to different sampling methods.

Token Selection Techniques:

- **Temperature**:
  - Controls randomness in token selection.
  - Lower temperature (e.g., 0.2) makes the model more **deterministic**, focusing on high-probability tokens, e.g. "blue" above.
  - Higher temperature (e.g., 1.0 or above) increases **diversity** by considering lower-probability tokens.
- **Greedy Sampling**:
  - Always selects the token with the highest probability.
  - Leads to predictable but potentially repetitive text.
- **Top-k Sampling**:
  - Limits token selection to the top k tokens with the highest probabilities.
  - Adds diversity by excluding low-probability tokens while retaining the most likely options.

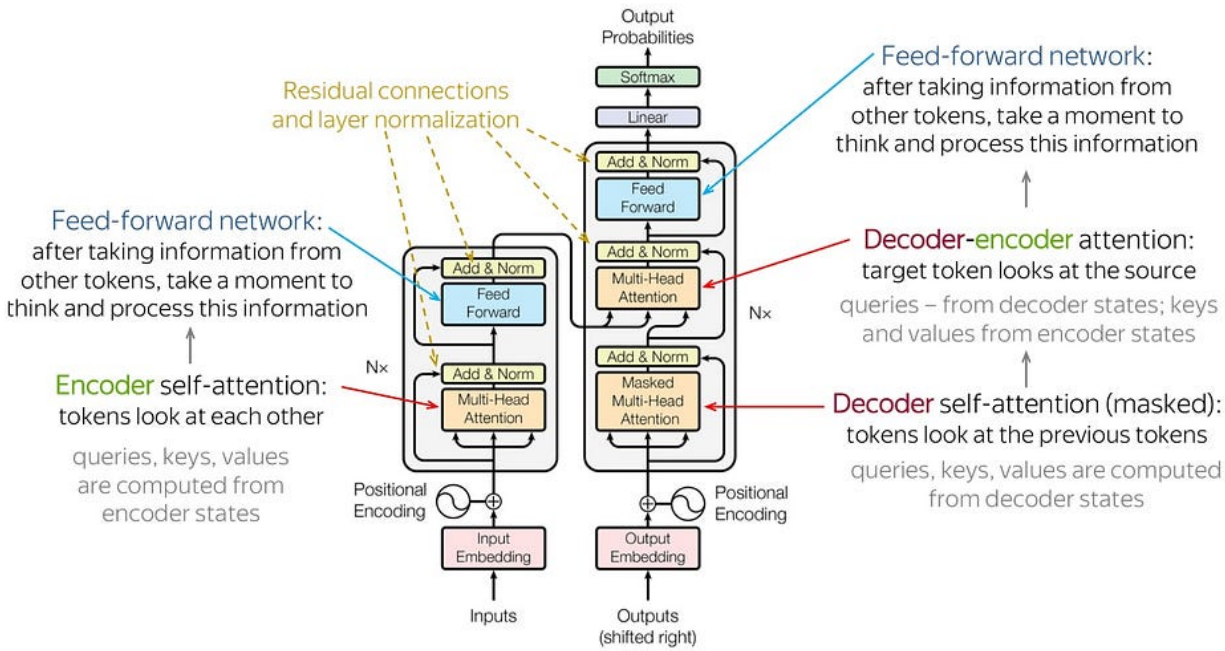
LLMs Critques

LMR show some downsides of LLMs when adopted in Economics (versions until 2024): 1) An LLM trained on "A is B" will not know "B is A."; 2) Minor changes in math: LLM solves (9/5)x+32 but cannot solve (7/5)x+31; 3) LLMs struggle on "counterfactual" versions of tasks ...

A more formal explanation of LLMs (more useful for the paper)

Transformers

How Transformers Work: A Step-by-Step Breakdown



Input Sequence

A sequence of words or tokens that the model will process.

Transformers

Transformers are architectures that significantly improved the performance of natural language tasks compared to earlier models like Recurrent Neural Networks (RNNs). Its key advantage lies in its ability to understand the relevance and context of all words in a sentence, not just neighboring words.

A complete transformers architecture (we consider this) is divided into:

- **Encoder:** Self-Attention + FFN
- **Decoder:** Masked self-attention + Self Attention + FFN + Softmax $\rightarrow$ otuput are probabilities as in Tab.1

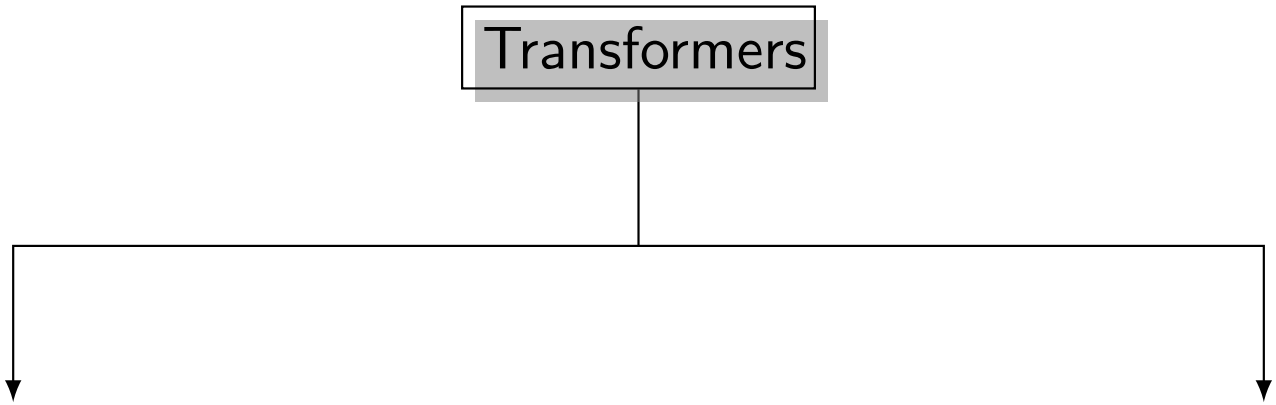
Notable exceptions:

- **BERT:** Encoder only: not goo for new text generation, but goo for classification;
- **(Old) GPTs:** Decoder only but used as input some pre-trained embeddings, which are generated by an encoder.

Since this is an introduction we won't go to all single aspects but we will focus on the main components common in encoders and decoders: Self-Attention and Feed Forward Neural Net (FFN).

Time allowing I will briefly give an intuition of the roles of Encoders and decoders along the way.

The macro-structure of Encoders and Decoders

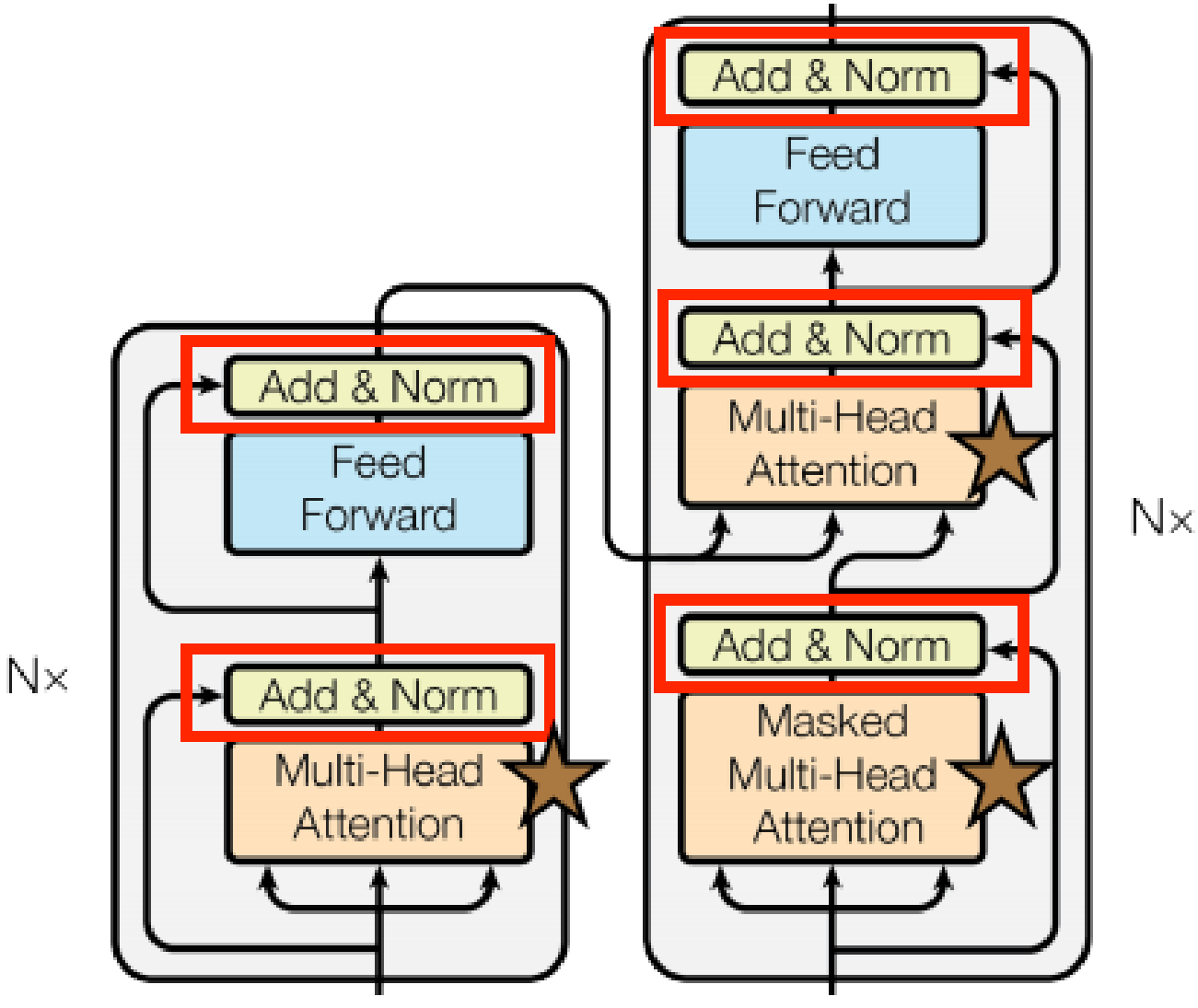


1) **Attention Layer:** $\longrightarrow$ 2) **Feed Forward Neural Net. (FFN):**

- (Pre-step) Input preparation: Tokenization + Embedding
- Context learning: self-attention
- Meaning: Positional-encoding
- Final output: a huge elaborated vector

- Refinement of transformer(s) result
- Normalization into probabilities
- Final output: Complex vector (encoder) or **probabilities** of different words that could follow the input token (decoder).

Attention Layer (starred)



Tokenization: transforming raw data into something the model can understand. E.g.: ("The cat sat on the mat"):

- *By word*: [The, cat, sat, on, the, mat]
- *By subword*: often the same here, but splits prefixes/suffixes when needed
- *By character*: [T, h, e, ␣, c, a, t, . . .]

Step 2. **Embedding:**

$$\text{Embed}: \ x \in \mathbb{R}^n \rightarrow y \in \mathbb{R}^m$$

- x: input token (e.g. "cat") y: corresponding numerical vector
- n: input dimension m: embedding dimension

Result after Step 2: $cat \rightarrow [0.25, 0.78, 0.45, \dots]$: language model is represented as a point in a multi-dimensional space.

Self-Attention (is all you need...)

Self-attention helps the model determine which words (or tokens) in a sentence are most relevant to each other when generating responses.

- When we read, we look back at earlier words to keep the context straight.
- In "The cat chased the mouse because it was hungry," it refers to the cat—we figure that out by linking words.
- Self-attention lets a model do the same: it looks at all the words at once and gives more weight to the ones most relevant to each word.
- Result: the model keeps track of who/what refers to what and which words matter most for meaning.

What is Attention concretely? A score attached to every token ad constructed as follows:

What is Attention concretely? A score attached to every token ad constructed as follows:

- **Input**: Convert each token to an embedding vector (e.g., "The cat chased the mouse" $\rightarrow$ vectors).

$\downarrow$

What is Attention concretely? A score attached to every token ad constructed as follows:

- **Input**: Convert each token to an embedding vector (e.g., "The cat chased the mouse" $\rightarrow$ vectors).

$\downarrow$

- **Q/K/V**: For each token, form **Query (Q)** (repres. of word we are focusing on), **Key (K) (repres. of all other words)**, **Value (V) (information we want to keep from each word)** vectors. These are used for comparing each word to every other word in sentence.

$\downarrow$

What is Attention concretely? A score attached to every token ad constructed as follows:

- **Input**: Convert each token to an embedding vector (e.g., "The cat chased the mouse" $\rightarrow$ vectors).

$\downarrow$

- **Q/K/V**: For each token, form **Query (Q)** (repres. of word we are focusing on), **Key (K) (repres. of all other words)**, **Value (V) (information we want to keep from each word)** vectors. These are used for comparing each word to every other word in sentence.

$\downarrow$

- **Attention Scores**: Relevance of token i to others $=$ dot products of its Q with all K's. Idea: measure of how relevant each word is to the query word;

$\downarrow$

What is Attention concretely? A score attached to every token ad constructed as follows:

- **Input**: Convert each token to an embedding vector (e.g., "The cat chased the mouse" $\rightarrow$ vectors).

$\downarrow$

- **Q/K/V**: For each token, form **Query (Q)** (repres. of word we are focusing on), **Key (K) (repres. of all other words)**, **Value (V) (information we want to keep from each word)** vectors. These are used for comparing each word to every other word in sentence.

$\downarrow$

- **Attention Scores**: Relevance of token i to others $=$ dot products of its Q with all K's. Idea: measure of how relevant each word is to the query word;

$\downarrow$

- **Softmax**: Convert scores into a probability distribution (weights sum to 1). Why? So that model can focus more on the most relevant words

↓

- **Weighted sum**:Each word's value V is multiplied by its attention score, and the results are summed to create a new representation of the word.

↓

Steps 1-5 can be summarized in a formula:

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

where

- Q: queries, K: keys, V: values
- d_k: key dimension (for scaling)

Notice: Self-attention is **linear** in V: with Q, K fixed: linear, the attention weights $A = \text{softmax}(\cdot)$ the map AV is linear.
Example: Representation for "it" in "[“The”, “cat”, “chased”, “the”, “mouse”, “because”, “it”, “was”, “hungry”]": (i). Calculate Attention Scores via formula: The model computes how well the query “it” relates to each of the keys (K) from the other words; (ii) The word “cat” would likely receive a higher score than “mouse” because “it” refers back to “cat.” (iii) The resulting representation for "it" would be a weighted sum of the value embeddings, emphasizing the context provided by the word “cat.”

For picky ones...

Say our sentence is "a b c D";

- Turn those words into tokens (each token is a 3-dimensional vector):

$$a = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}, \quad b = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}, \quad c = \begin{bmatrix} c_1 & c_2 & c_3 \end{bmatrix}, \quad D = \begin{bmatrix} D_1 & D_2 & D_3 \end{bmatrix}.$$

- Stack the 4 tokens into a matrix X (shape 4×3):

$$X = \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \\ D_1 & D_2 & D_3 \end{bmatrix} \in \mathbb{R}^{4\times 3}.$$

- **How are we sure that Attention divides tokens into Q,K,V s.t. we can learn context? Key: learned weights.** To prepare for attention, we must first generate the K, Q, and V using **LEARNED** weighted matrices. For this sentence, we want to transform it into a 4×2 matrix. So, each of the weight matrices will be of shape 3×2. For example, below is the weight matrix for Q named W:

$$W = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \\ Q_{31} & Q_{32} \end{bmatrix} \in \mathbb{R}^{3\times 2}.$$

Using W, obtain the query matrix:

$$Q = X\,W \in \mathbb{R}^{4\times 2} = \begin{bmatrix} a_1Q_{11} + a_2Q_{21} + a_3Q_{31} & a_1Q_{12} + a_2Q_{22} + a_3Q_{32} \\ b_1Q_{11} + b_2Q_{21} + b_3Q_{31} & b_1Q_{12} + b_2Q_{22} + b_3Q_{32} \\ c_1Q_{11} + c_2Q_{21} + c_3Q_{31} & c_1Q_{12} + c_2Q_{22} + c_3Q_{32} \\ D_1Q_{11} + D_2Q_{21} + D_3Q_{31} & D_1Q_{12} + D_2Q_{22} + D_3Q_{32} \end{bmatrix}$$

$$= \begin{bmatrix} a_1^Q & a_2^Q \\ b_1^Q & b_2^Q \\ c_1^Q & c_2^Q \\ D_1^Q & D_2^Q \end{bmatrix} = \begin{bmatrix} a^Q \\ b^Q \\ c^Q \\ D^Q \end{bmatrix},$$

- **Why 4×2?** 4×2 is a design choice, not a mathematical necessity:
  - $4 =$ sequence length. We have 4 tokens (a, b, c, D), so the first dimension stays 4 no matter what. Attention keeps one row per token.
  - $2 =$ head dimension d_k. The second dimension is chosen by the model designer (or the model per se if using an OpenAI architecture for instance) for the size of the query/key (and often value) vectors. You could pick 1, 2, 3, 4, ...—it's a hyperparameter.
- Notice how each vector in the resulting matrix Q is not a linear combination of all other tokens! Rather, each vector is a linear combination of itself and some weights. The first vector is just a linear combination of "a". The second is just a linear combination of "b"...
- This transformation does not mess up the sequence order within the matrix. a is still at the top of the matrix and D is still at the bottom of the matrix.
- **Same process for forming K and V.**

NOTE:

Modern LLMs, use **Multi-head self-attention** to capture different aspects of language.

Each head learns a different relationship, such as identifying entities, actions, or other properties (learned in the pre-training on the corpus: remember categories like "colour"?).

Example: "She saw the moon with a telescope": one head might focus on the subject ("she"), another on the action ("saw"), and another on the object ("telescope").

Objective: being faster.

What about Position?

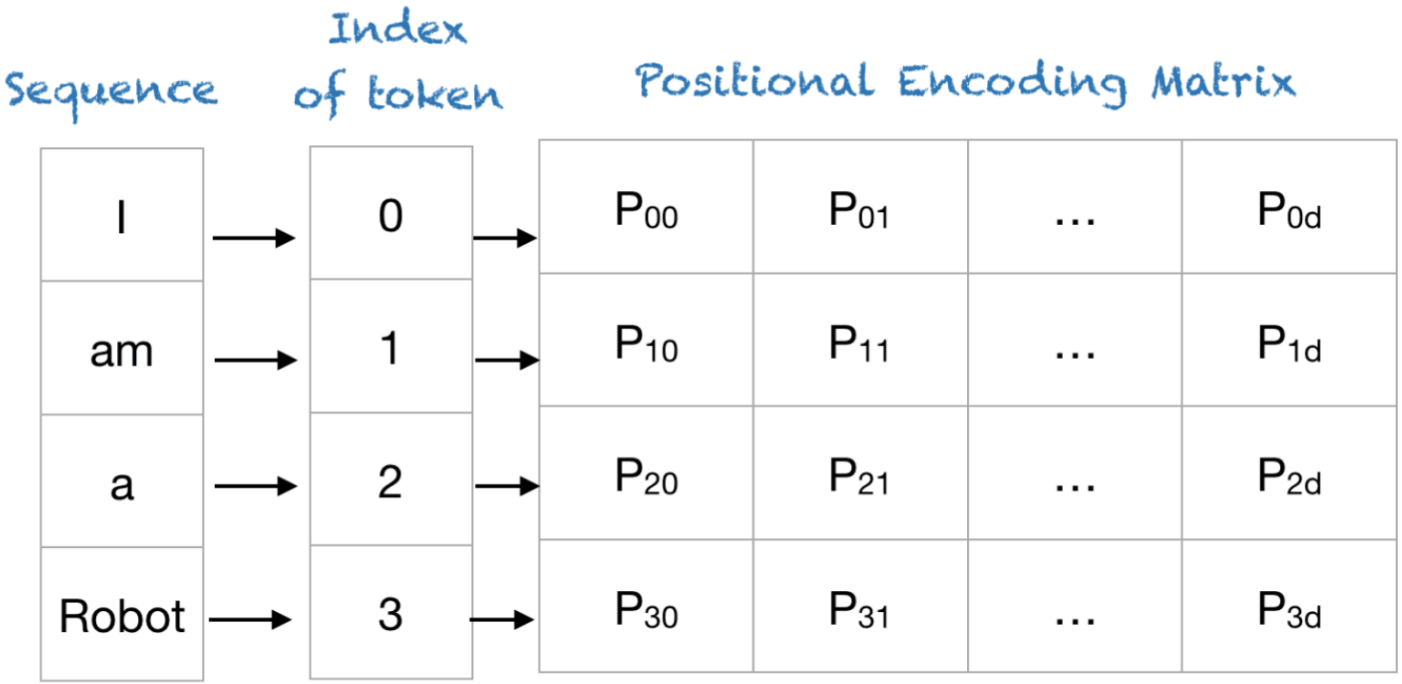
Problem: Transformer architecture, treats all words simultaneously without any sense of order. .Attention does not say anything about the word ordering in a sentence! Ordering is crucial for the meaning of the sentence, e.g. *"The cat chased the mouse"* and *"The mouse chased the cat"* have completely different meanings despite using the same words.

Solution: Step 3. **Positional Encoding**. Give the model information about the position of each token in a sequence. **How?** Adds specific numerical values to the token embeddings based on their position in the sentence.

Why not just a single number in embedding indexing position? After all, we could think of simply adding a further dimension and put a number indexing the position of the token in the sequence!

The latter is a **bad idea** for several reasons. The more intuitive is that for long sequences, the indices can grow large in magnitude. If you normalize the index value to lie between 0 and 1, it can create problems for variable length sequences as they would be normalized differently!

So what should we do? Intuitively, transformers use a smart positional encoding scheme, where each position/index is mapped to a **unique** vector. Hence, the output of the positional encoding layer is a matrix, where each row of the matrix represents an encoded object of the sequence summed with its positional information. (see next slide)



Positional Encoding Matrix for the sequence 'I am a robot'

Sentence of length L and require position of k^{th} element:

$$\mathrm{PE}(k, 2i) = \sin\left(\frac{k}{10000^{\frac{2i}{d_{\text{model}}}}}\right), \qquad \mathrm{PE}(k, 2i+1) = \cos\left(\frac{k}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

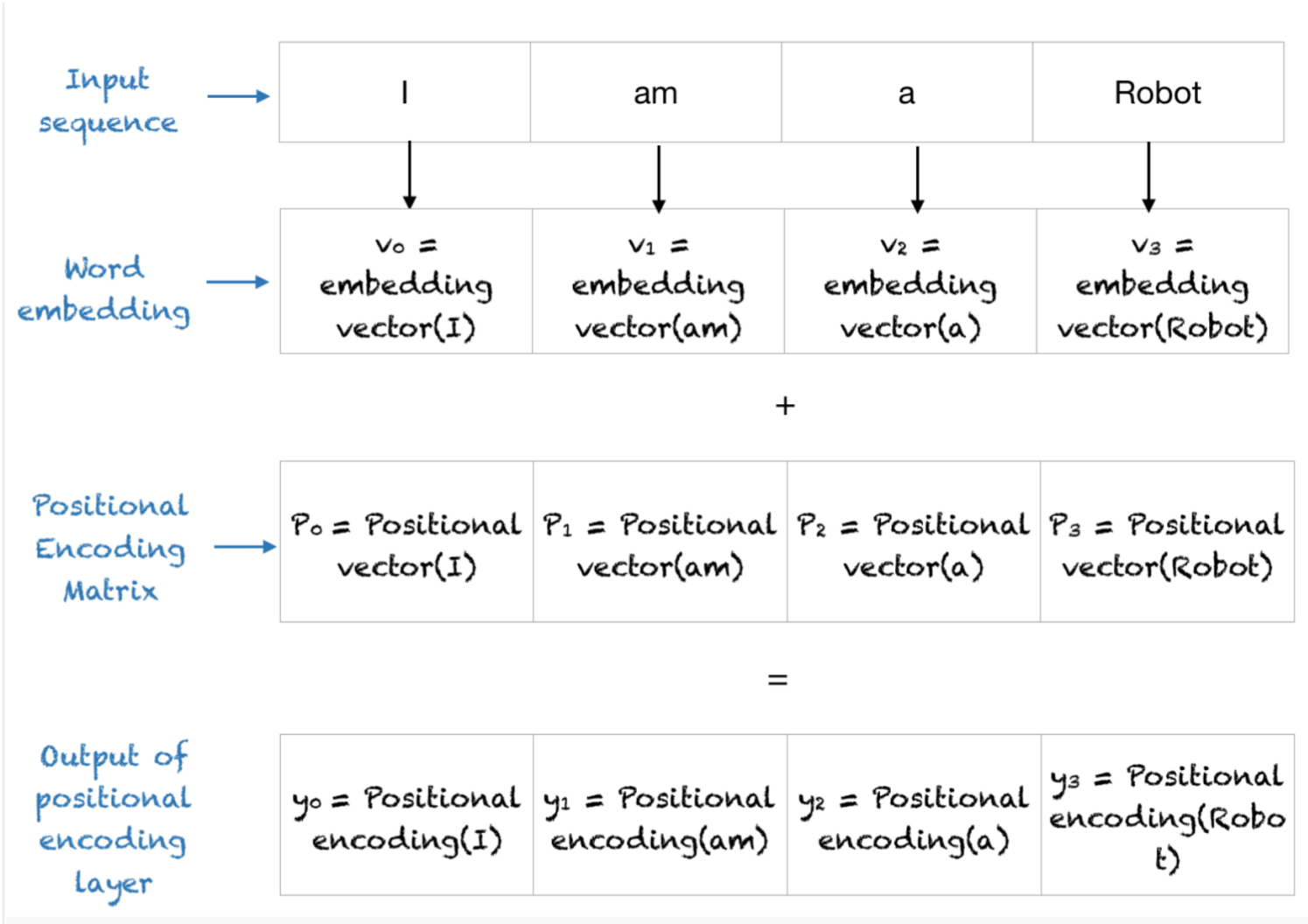
- $PE(k, j)$ Position function for mapping a position k in the input sequence to index $((k, j))$ of the positional matrix
- i: Used for mapping to column indices $0 \leq i < d_{model}/2$, with a single value of i maps to both sine and cosine functions
- k: Position of an object in the input sequence, $0 \leq k < L/2$ (e.g. $0, 1, 2, \cdots$)
- d_{model}: total embedding dimensionality.
- **Why sine & cosine?** They create smooth, repeating waves at multiple frequencies, so nearby positions have similar codes and the model can sense both short- and long-range distances: sin for even and cos for odd positions in matrix.

Sequence	Index of token, k	Positional Encoding Matrix with d=4, n=100			
		i=0	i=0	i=1	i=1
I	0	P_{00}=sin(0) = 0	P_{01}=cos(0) = 1	P_{02}=sin(0) = 0	P_{03}=cos(0) = 1
am	1	P_{10}=sin(1/1) = 0.84	P_{11}=cos(1/1) = 0.54	P_{12}=sin(1/10) = 0.10	P_{13}=cos(1/10) = 1.0
a	2	P_{20}=sin(2/1) = 0.91	P_{21}=cos(2/1) = -0.42	P_{22}=sin(2/10) = 0.20	P_{23}=cos(2/10) = 0.98
Robot	3	P_{30}=sin(3/1) = 0.14	P_{31}=cos(3/1) = -0.99	P_{32}=sin(3/10) = 0.30	P_{33}=cos(3/10) = 0.96

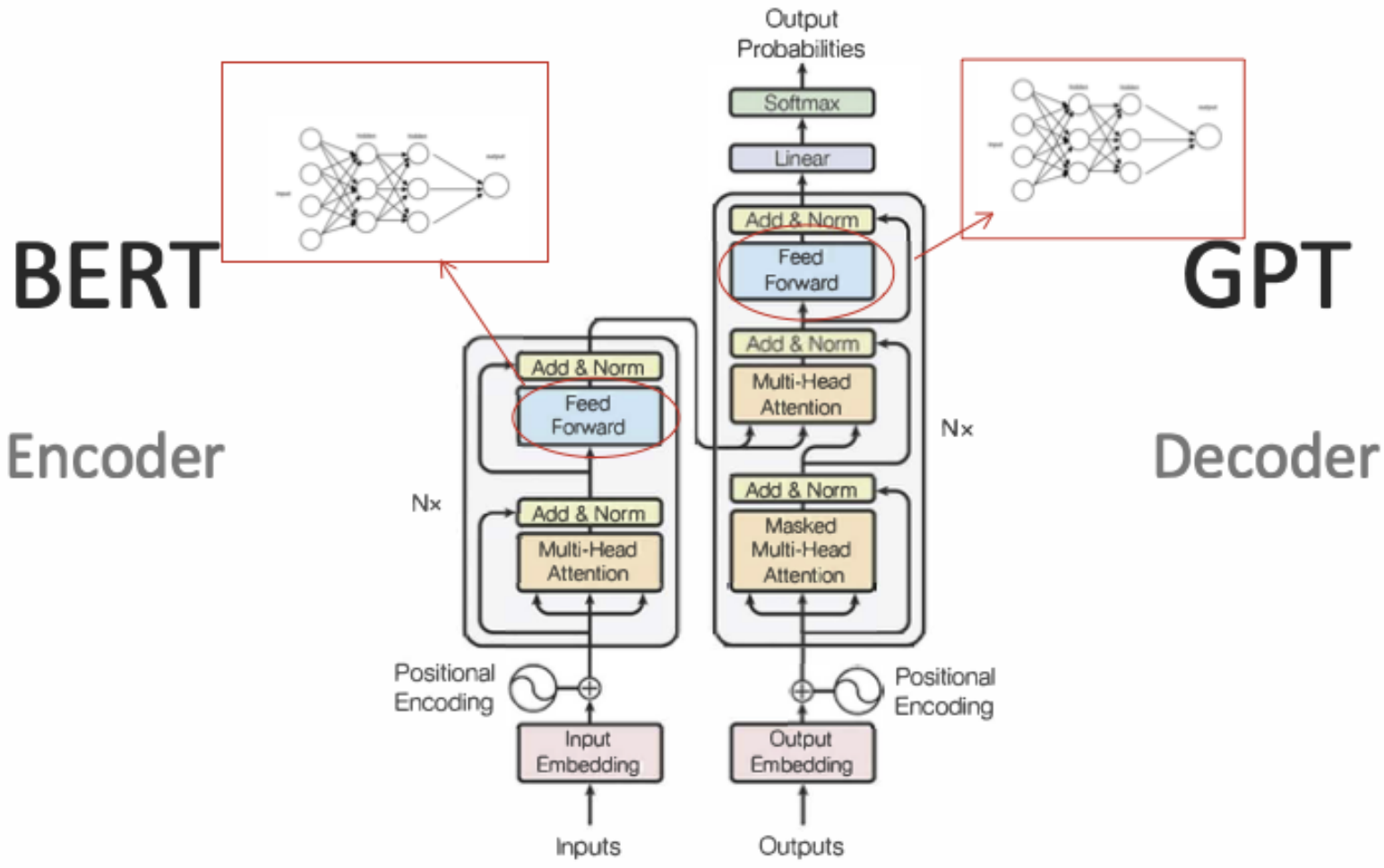
Positional Encoding Matrix for the sequence 'I am a robot'

- 10000 is a user chosen number
- **Relative distance emerges:** Phase shifts of sin/cos make offsets ("how far apart") easy to learn via dot products in attention (e.g. cosine-similarity idea applied to attention).
- **Same size as Step 2. embeddings:** $\mathrm{PE}(k) \in \mathbb{R}^{d_{\text{model}}}$ so we can add it **elementwise** to the token embedding—no shape changes—and every feature channel carries position info.
- **Effect:** positional encoding is added to its corresponding embedding vector. This combined vector allows the model to consider both the word's meaning and its position in the sentence.

WHAT IS THE OUTPUT OF POSITIONAL ENCODING LOOK LIKE?

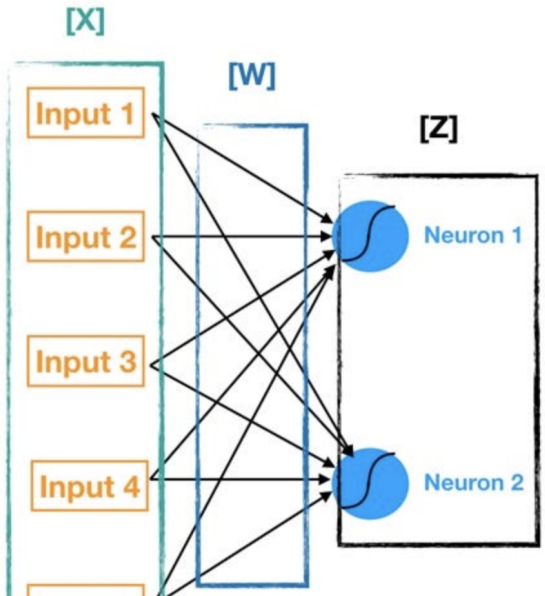


FFN


BERT
Encoder
GPT
Decoder
Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs

How do signals move in NN?

- **Weighted input:** Each connection between neurons has a weight, which determines the strength of the signal passing through it.
- **Summation:** A neuron receives multiple weighted inputs from other neurons. These inputs are summed together at a summation junction. So $[W] \otimes [X] + [Bias] = [Z]$
- **Activation function:** The summed signal is then passed through an activation function to produce the neuron's output signal. This step introduces a non-linearity;
- **Layered movement:** This process repeats across multiple layers. The output from a neuron in one layer becomes the input for neurons in the next layer.



FFN is a series of layers (input layer + one or more hidden layers + output layer. No inner loops.) stacked together: each layer transforms the input data from preceeding layer into a new representation.

1. Input layer receives the output representations from the attention mechanism;
2. Hidden Layers: each hidden layer consists of multiple neurons. Each neuron receives input from the previous layer, applies a weight to it, adds a bias, and then passes the result through a ReLU activation function;
3. Output Layer: produces the results (probabilities for different words).

In one line, 1-3 are:

$$\mathbf{h} = \mathrm{ReLU}\Big(\mathbf{W}_2 \cdot \mathrm{ReLU}\overbrace{(\mathbf{W}_1 \cdot \mathbf{x} + \mathbf{b}_1)}^{\text{Output of previous layer}} + \mathbf{b}_2\Big).$$

where $\mathbf{h}$ is the FFN's output; $\mathbf{W}_1$ and $\mathbf{W}_2$ are weight matrices connecting the layers; $\mathbf{b}_1$ and $\mathbf{b}_2$ are bias vectors; and $\mathrm{ReLU}(z) = \max\{0, z\}$ is the activation that introduces nonlinearity, allowing the model to capture more complex patterns.

Activation functions are important to add non-linearities in the NN: they transform the input signal of a node into an output signal that is then passed on to the next layer.

Without activation functions, neural networks would be restricted to modeling only linear relationships (e.g. $h_k = W_k \cdot z_{k-1} + b$ rather than $h_k = g(W_k \cdot x + b)$) between inputs and outputs.

Why introducing non-linearities is important? Most real-world data is non-linear $\rightarrow$ intuitively if we were forced to use linear relationships we would need more layers and nodes to reconstruct potential non-linearities in data (e.g. construct a circle using only lines requires a lot of linear equations vs constructing it using second order equations requires only one!)

How can something this simple $\mathrm{ReLU}(z) = \max\{0, z\}$ **allows for a lot of non-linearities?** Imagine this as a lego problem: ReLU gives you the possibility of combining more non-linear shapes rather than just linear one (check graph of ReLU and imagine to add it/modify it at every layer...you pretty much can reconstruct every function as a sum of ReLUs)...

Example

- "cat" = [0.23,0.14,0.11,0.46]
- Normalization step to avoid gradient vanishing in layers (see pag. 39): $x = \frac{cat-\mu}{\sigma}$
- First Layer: $z_1 = W_1 \cdot x + b_1$
- Activation: $h_1 = ReLU(z_1)$
- Output: Probabilities of different words that could follow the input token "cat".

How do we understand if the output probabilities are correct? LLMs learn in the training phase by adjusting their internal parameters during training.

But how do they know whether they're doing well or need improvement? This is where the **loss function** (typically cross-entropy) comes in. The loss function measures how far off the model's predictions are from the actual correct answers, helping it to "learn" and improve over time.

Once these parameters are learned (loss almost 0) the model is trained and can learn on new sequences (which is what we do when PROMPTING an LLM!).

So, in the training phase the model has learned some loss-minimizing parameters...but **how to ensure outer-validity?** (i.e. that predictions are loss minimizing also off sample)?

We need a way to improve the model's performance. This is done through a process called **backpropagation** combined with optimization. These methods adjust the model's internal parameters so that it makes better predictions in the future.

Backpropagation

- Forward Pass: The model makes a prediction by passing the input data through all the layers of the network.
- Calculate Loss: The loss function calculates how far off the model's prediction was from the correct answer.
- **Backpropagation**: The model calculates the **gradient** of the loss function wrt each weight. This tells us **how much a small change in each weight will affect the loss (i.e. how much the model is robust to out-of-sample prompts)** $\rightarrow$ if a slight change in W completely messes the output, the model is unreliable!
- Update Weights: Using these gradients, the model updates its weights to reduce the error, a process called optimization.

Example:

- *Forward:* Suppose the model predicts the next word in a sentence and makes a mistake. The true word is "mat", but the model predicts "chair" with the highest probability..
- *Loss:* The cross-entropy loss is calculated, and it turns out to be relatively high because the model predicted the wrong word.
- *Backward:* The model calculates the gradient of the loss function with respect to each weight. It identifies that some weights contributed more to the error than others.
- *Update:* Using the gradients, the model adjusts its weights. For example, the weight responsible for predicting "chair" might be decreased, while the weight for "mat" is increased. How? SGD/Adam step $\theta \leftarrow \theta - \eta \nabla_\theta L$. E.g., if $\theta_0 = 0.5$, $\frac{\partial L}{\partial \theta} = -0.1$, $\eta = 0.05$, then $\theta_1 = 0.5 - 0.05(-0.1) = 0.505$ (increasing the weight helps raise $p_\theta(\text{"mat"} \mid x)$).
- *Repeat:* iterate over many examples to improve predictions.

The gradient descend problem (intuition)

Denote hidden states $h_1, h_2, \ldots$, inputs $u_1, u_2, \ldots$, and outputs $x_1, x_2, \ldots$. Let it be parameterized by θ, so that the system evolves as

$$(h_t, x_t) \;=\; F(h_{t-1}, u_t, \theta).$$

Simplify to the case where $x_t = h_t$ (e.g. one layer) since the problem already presents here:

$$x_t \;=\; F(x_{t-1}, u_t, \theta).$$

Now, take the differential (over all dimensions):

$$\begin{aligned}
dx_t &= \nabla_\theta F(x_{t-1}, u_t, \theta)\, d\theta + \nabla_x F(x_{t-1}, u_t, \theta)\, dx_{t-1} \\
&= \nabla_\theta F(x_{t-1}, u_t, \theta)\, d\theta + \nabla_x F(x_{t-1}, u_t, \theta)[\nabla_\theta F(x_{t-2}, u_{t-1}, \theta)\, d\theta + \\
&+ \nabla_x F(x_{t-2}, u_{t-1}, \theta)\, dx_{t-2} \\
&\;\;\vdots \\
&= \Big[\nabla_\theta F(x_{t-1}, u_t, \theta) + \nabla_x F(x_{t-1}, u_t, \theta)\, \nabla_\theta F(x_{t-2}, u_{t-1}, \theta) + \cdots\Big]\, d\theta.
\end{aligned}$$

where each substitution is by recursion (e.g. substitute dx_{t-x} using the expression of dx_t above but in $t-1$...)

Training requires a loss function to minimize; let loss be $L = L(x_T, u_1, \ldots, u_T)$. Then gradient descent gives

$$dL = \nabla_x L(x_T, u_1, \ldots, u_T)\, dx_t =$$
$$= \nabla_x L(x_T, u_1, \ldots, u_T) \Big[\nabla_\theta F(x_{t-1}, u_t, \theta) + \nabla_x F(x_{t-1}, u_t, \theta)\, \nabla_\theta F(x_{t-2}, u_{t-1}, \theta)$$
$$+ \cdots \Big]\, d\theta,$$

and the parameter update is

$$\Delta\theta = -\eta \Big[\nabla_x L(x_T) \Big(\nabla_\theta F(x_{t-1}, u_t, \theta)$$
$$+ \nabla_x F(x_{t-1}, u_t, \theta)\, \nabla_\theta F(x_{t-2}, u_{t-1}, \theta) + \cdots \Big) \Big]^\top,$$

where η is the learning rate.
The vanishing/exploding gradient problem appears because of repeated Jacobian multiplications of the form

$$\nabla_x F(x_{t-1}, u_t, \theta)\, \nabla_x F(x_{t-2}, u_{t-1}, \theta)\, \nabla_x F(x_{t-3}, u_{t-2}, \theta) \cdots$$

which can shrink or blow up the gradient norm over long time horizons.

So **why is this concretely an issue?**

Example with RNN with sigmoid activation: Consider a typical recurrent network

$$x_t \;=\; F(x_{t-1}, u_t, \theta) \;=\; W_{\mathrm{rec}}\,\sigma(x_{t-1}) \;+\; W_{\mathrm{in}}\,u_t \;+\; b,$$

where $\theta = (W_{\mathrm{rec}}, W_{\mathrm{in}})$, σ is the (element wise) sigmoid, and b is a bias vector. The Jacobian w.r.t. x at time t is

$$\nabla_x F(x_{t-1}, u_t, \theta) \;=\; W_{\mathrm{rec}}\,\mathrm{diag}\Big(\sigma'(x_{t-1})\Big).$$

Hence a k-step Jacobian product looks like

$$\begin{aligned}
&\nabla_x F(x_{t-1}, u_t, \theta)\;\nabla_x F(x_{t-2}, u_{t-1}, \theta)\;\cdots\;\nabla_x F(x_{t-k}, u_{t-k+1}, \theta)\\
&\qquad=\; W_{\mathrm{rec}}\,\mathrm{diag}\Big(\sigma'(x_{t-1})\Big)\;W_{\mathrm{rec}}\,\mathrm{diag}\Big(\sigma'(x_{t-2})\Big)\;\cdots\;W_{\mathrm{rec}}\,\mathrm{diag}\Big(\sigma'(x_{t-k})\Big).
\end{aligned}$$

Since $|\sigma'(z)| \le 1$ for all z, the operator norm of the above product is bounded by $\|W_{\mathrm{rec}}\|^k$. If the spectral radius of W_{rec} is $\gamma < 1$, then for large k the norm is bounded by $\gamma^{\,k} \to 0$. This is the prototypical vanishing-gradient phenomenon.

The effect on the loss gradient is seen from

$$\nabla_\theta L = \nabla_x L(x_T, u_1, \ldots, u_T) \Big[\nabla_\theta F(x_{t-1}, u_t, \theta) + \nabla_x F(x_{t-1}, u_t, \theta) \nabla_\theta F(x_{t-2}, u_{t-1}, \theta) + \cdots \Big].$$

If $\|\nabla_x F(x_{t-k}, u_{t-k+1}, \theta)\| \lesssim \gamma^k$ with $\gamma < 1$ and $\nabla_\theta F$ is bounded by some $M > 0$, then the k-step terms in $\nabla_\theta L$ decay like $M\,\gamma^k$. Effectively, only the first $O(\gamma^{-1})$ terms contribute appreciably, and very long-range effects are lost. (If $\gamma \geq 1$, the above bound no longer implies decay; this is related to the exploding-gradient case.)

- Each time step multiplies the backpropagated signal by something whose size is less than 1 (because sigmoid derivatives are ≤ 1 and the recurrent weight matrix has spectral radius $\gamma < 1$).
- Multiplying numbers < 1 many times makes the signal shrink exponentially.
- As a result, events that happened many steps in the past barely change the loss, so the model gets almost no learning signal to adjust weights based on long-range dependencies.
- Practically: the RNN forgets long-term information and mainly learns short-term patterns; early time steps don't get trained effectively.

Long story short

FFN is usually adopted after the Self-Attention mechanism(s). Depending on whether it is in encoder or decoder, it takes as input the "interpreted by Self-Attention" vectors and outputs probabilities over a vocabulary.

Attention to vanishing gradient $\rightarrow$ i.e. normalize inputs of each layer.

Why I pointed it out? Because if you build an LLM or train a transformer it is important that you remember to normalize!

Go back to the Encoder-Decoder structure of Transformers to see the role of Self-Attention an FFN in context...

Encoder-Decoder Logic

How are these structures used in the Encoder-Decoder logic?

Encoder uses Self-Attention + FFN to prepare the information for the decoder. The encoder's output is a sequence of vectors, one for each word in the input sentence. Each vector now encapsulates information about the word itself, its position, and its relationships with other words.

Decoder generates the output sequence from the processed input. It uses similar layers to the encoder (2 x Self-Attention + FFN) but includes mechanisms to handle the previously generated tokens and focus on relevant input tokens.

FFN in Encoders

In decoder FFN just refines the output of Layers 1-2 (not special differences wrt Encoder's FFN). So we won't touch the FFN apart from the question below...

Why is the FFN needed in the encoder if it just outputs the probabilities of next word?
Encoders don't produce next-word probabilities; they produce contextual representations.
In this context FFN serves to further classify words an refine the attention layer: it exploits nonlinearities to capture patterns that the linear self-attention (linear in V, see above) did not get!

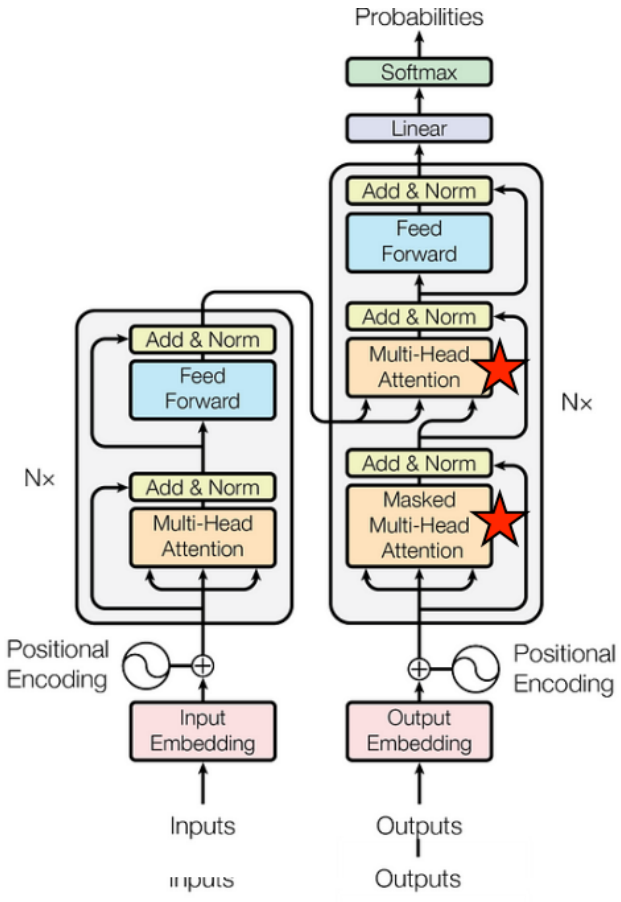
Take the sentence: "The river overflowed near the bank." After self-attention, the word "bank" pulls info from nearby words like "river" and "overflowed." So its vector now carries hints of water.
The FFN then takes that mixed vector and classifies new features: "bank = riverside place," "not a finance institution," "location entity," etc.

Because of the nonlinearity and learned weights, FFN, boosts the "river" sense features and suppresses "finance" sense (a decision you can't get by just linearly mixing neighbors which is done by self-attention.)

Self-Attention in Decoders

BERT

Encoder



GPT

Decoder

The Decoder **uses two layers of Self-Attention in a slight different way from Encoder**: Layer 1) used to decide how much importance to give each part of the input sentence while generating the next word in the sequence. Layer 2) Maintain autoregressive property in prediction. Sometimes inverted.

Aim of the decoder is to generate the next token one-by-one. This is done via Layer 1 + **masked** Layer 2 over the already generated (by the encoder) ad filtered (by Layer 1) tokens.

Mask serves as anti-cheating mechanism. The mask blocks future positions. Intuition: writing left to write: At each word, you can reread everything you've written so far and decide which parts matter most—but you can't see the future words yet.

An **attention mask** is a binary tensor that signals which **tokens** should be considered (with non-zero weights) and which should be ignored (with zero weights). Serves to maintain autoregressive property (see following).

So

- **WITHOUT** mask, self-attention: $\text{softmax}(QK^T)V$
- **WITH** mask, self-attention: $\text{softmax}(QK^T + M)V$ where M is the mask matrix:

$$M = \left[\begin{array}{c|cccc} & a^K & b^K & c^K & D^K \\ \hline a^Q & 0 & 0 & 0 & -\infty \\ b^Q & 0 & 0 & 0 & -\infty \\ c^Q & 0 & 0 & 0 & -\infty \\ D^Q & 0 & 0 & 0 & -\infty \end{array}\right]$$

- The look-ahead mask is used so that the model can be trained on an entire sequence of text at once as opposed to training the model one word at a time (to avoid looking at future)!
- Masking so the model cannot look ahead at future tokens, only past tokens!

How does the mask affect Attention? Example masking only "D" in " a b c D".

Let's adopt a (more convenient) vector notation for QK^T at pag.22:

$$QK^\top = \left[\begin{array}{c|cccc} & a^K & b^K & c^K & D^K \\ \hline a^Q & a^Q a^K & a^Q b^K & a^Q c^K & a^Q D^K \\ b^Q & b^Q a^K & b^Q b^K & b^Q c^K & b^Q D^K \\ c^Q & c^Q a^K & c^Q b^K & c^Q c^K & c^Q D^K \\ D^Q & D^Q a^K & D^Q b^K & D^Q c^K & D^Q D^K \end{array}\right]$$

Adding the mask QK^T we obtain:

$$QK^\top + M = \left[\begin{array}{c|cccc} & a^K & b^K & c^K & D^K \\ \hline a^Q & a^Q a^K & a^Q b^K & a^Q c^K & -\infty \\ b^Q & b^Q a^K & b^Q b^K & b^Q c^K & -\infty \\ c^Q & c^Q a^K & c^Q b^K & c^Q c^K & -\infty \\ D^Q & D^Q a^K & D^Q b^K & D^Q c^K & -\infty \end{array}\right]$$

So applying softmax $-\infty$ becomes 0:

$$\text{softmax}(QK^\top + M) = \left[\begin{array}{c|cccc} & a^K & b^K & c^K & D^K \\ \hline a^Q & (a^Q a^K)_S & (a^Q b^K)_S & (a^Q c^K)_S & 0 \\ b^Q & (b^Q a^K)_S & (b^Q b^K)_S & (b^Q c^K)_S & 0 \\ c^Q & (c^Q a^K)_S & (c^Q b^K)_S & (c^Q c^K)_S & 0 \\ D^Q & (D^Q a^K)_S & (D^Q b^K)_S & (D^Q c^K)_S & 0 \end{array}\right]$$

So D^K is all 0s now! We just learned a way to "hide the future"!

Remember indeed that the original sentence at pag.22 was "a b c D": the D^K component which we followed along the way has thus 0 effect on the Q, K, V layer. Indeed when looking at $\text{softmax}(QK^T)V$ we see that D component is 0:

$$\text{softmax}\left(QK^\top + M\right)V = \underbrace{\begin{bmatrix} (a^Q a^K)_S & (a^Q b^K)_S & (a^Q c^K)_S & 0 \\ (b^Q a^K)_S & (b^Q b^K)_S & (b^Q c^K)_S & 0 \\ (c^Q a^K)_S & (c^Q b^K)_S & (c^Q c^K)_S & 0 \\ (D^Q a^K)_S & (D^Q b^K)_S & (D^Q c^K)_S & 0 \end{bmatrix}}_{\text{masked, row-wise softmax}} \underbrace{\begin{bmatrix} a_1^V & a_2^V \\ b_1^V & b_2^V \\ c_1^V & c_2^V \\ D_1^V & D_2^V \end{bmatrix}}_{V}$$

$$= \begin{bmatrix} (a^Q a^K)_S\, a_1^V + (a^Q b^K)_S\, b_1^V + (a^Q c^K)_S\, c_1^V + 0 & (a^Q a^K)_S\, a_2^V + (a^Q b^K)_S\, b_2^V + (a^Q c^K)_S\, c_2^V + 0 \\ (b^Q a^K)_S\, a_1^V + (b^Q b^K)_S\, b_1^V + (b^Q c^K)_S\, c_1^V + 0 & (b^Q a^K)_S\, a_2^V + (b^Q b^K)_S\, b_2^V + (b^Q c^K)_S\, c_2^V + 0 \\ (c^Q a^K)_S\, a_1^V + (c^Q b^K)_S\, b_1^V + (c^Q c^K)_S\, c_1^V + 0 & (c^Q a^K)_S\, a_2^V + (c^Q b^K)_S\, b_2^V + (c^Q c^K)_S\, c_2^V + 0 \\ (D^Q a^K)_S\, a_1^V + (D^Q b^K)_S\, b_1^V + (D^Q c^K)_S\, c_1^V + 0 & (D^Q a^K)_S\, a_2^V + (D^Q b^K)_S\, b_2^V + (D^Q c^K)_S\, c_2^V + 0 \end{bmatrix}.$$

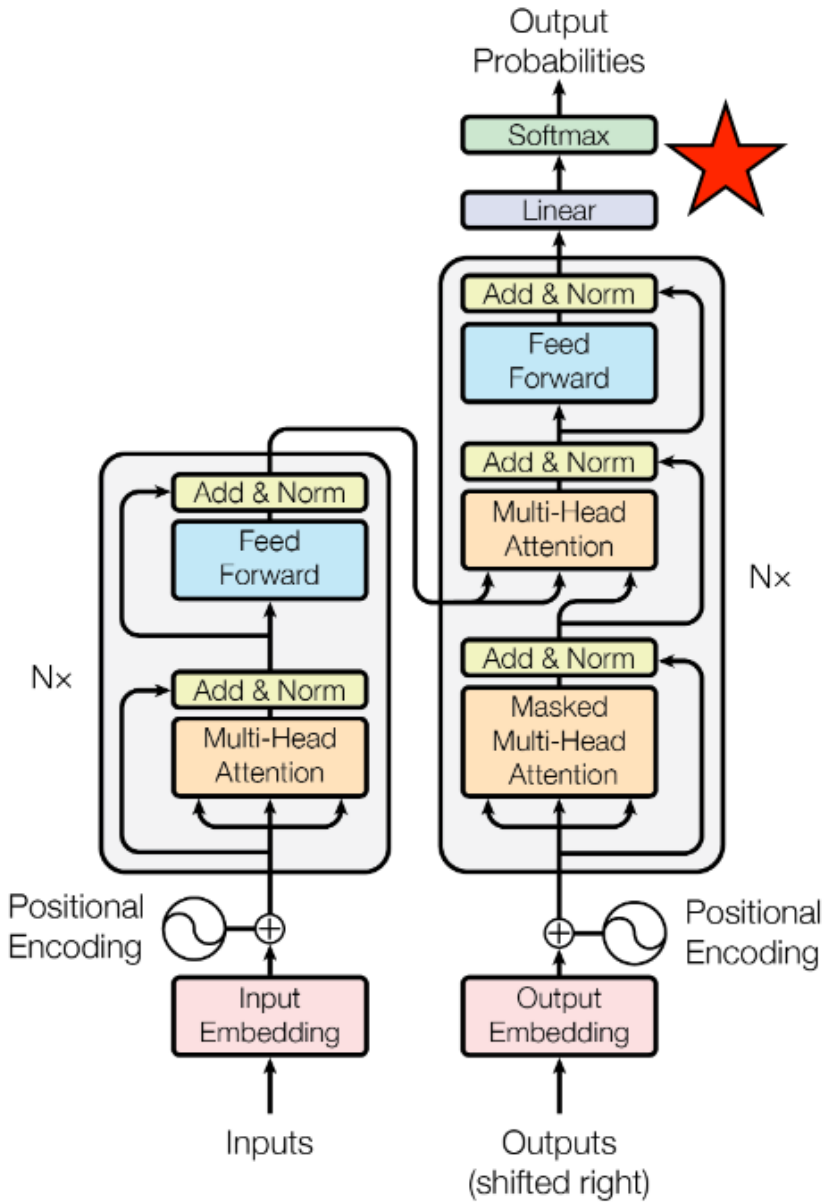
However "future" is relative: for first word, the future are all the next $n-1$ words (e.g. "b c D" for "a b c D"), for the second the next $n-2$ and so on...

How to design a mask matrix to account for this?

$$\mathrm{M} = \left[\begin{array}{c|cccc} & a^K & b^K & c^K & D^K \\ \hline a^Q & 0 & -\infty & -\infty & -\infty \\ b^Q & 0 & 0 & -\infty & -\infty \\ c^Q & 0 & 0 & 0 & -\infty \\ D^Q & 0 & 0 & 0 & 0 \end{array}\right]$$

Summary:
Future is typically masked in decoders: preserves the autoregressive rule "predict next token from the past only." $\rightarrow$ which "past"? The more important base on Attention $\rightarrow$ which Attention? The Attention derived by the input (encoder given) and filtered by Layer 1 attention of decoder (emphasize the most important parts for the reply)

The end of Transformers



The output from the two attention layers (and FFN after) of the decoder are logits. **Logits** are unnormalized, raw values generated by the model for each possible next token before they are converted into probabilities using Softmax.

Softmax function converts the raw output scores (logits) into a probability distribution over the vocabulary of the LLM. Output is like Table 1 a most likely next word (given context ad meaning of sentences as seen in the whole transformer) is chosen.

But how can ChatGPT form sentences if it only predicts the next word? Avalanche (or chain)mechanism! It is predicting the next word and using the new sentence as an input for the transformer to predict the next one and so on: e.g. initial input sentence $s \rightarrow$ predict $w1 \rightarrow$ use $s_1 = s + w1$ as new input to predict $w2$ and form $s_2 = s_1 + w2 \rightarrow$ use s_2 as new input to predict $w3$...

Space for randomness

We said that the output of decoder are some logits, and, for simplicity, only mentioned that the decision taken by the decoder at final step is stochastic. However, this is important to answer questions like: "do we expect the same output if we input repeatedly the same prompt?"

In principle, **the decoder has a categorical distribution over tokens** $p(i|context)$ possibly reshaped by some parameters like temperature, greedy samp., top-k.

The decoder, when "choosing which will be the next word" then chooses one token (**by sampling or greedy argmax**) and appends it.

Thus, **with the same exact prompt and the same decoding settings:**

- If "greedy" is on: we always obtain the same next word, i.e. the most likely one
- If "sampling" is on repeated runs can yield different next words. As we will see, which replies will be more likely depends on temperature T: e.g. if $T \to 0$ we will obtain the most likely (next) word for each sample (the same as greedy) **most of the times but not always!**

So summarizing:

- The distribution $p(i|context)$ is fully determined by the prompt and the model;
- Sampling means "changing the seed" at every iteration (keeping prompt and temperature fixed), i.e. influences the way the decoder picks from $p(i|context)$. Same as in data analysis!
- Example: Suppose after your prompt the next-token probs are [p("Engeneer")=0.5, p("Teacher")=0.1, p("Data Analyst")=0.4]. Seed 123 $\rightarrow$ draws Engineer. Seed 456 $\rightarrow$ PRNG draws Data Analyst.
  The distribution (p) didn't change; only the random pick from p did.

Why useful to try different seeds? E.g. if there are more "very likely" next words we need to know + robustness!

Focus on: Sampling

Sampling over which distribution? The **Model-distribution is different from the true** (real-world) distribution over alternatives (e.g. possible jobs). The **former can be learned, the latter is unknown**.
Model-distribution is a distribution over the **same** decoding parameters (temperature, top-k...) where randomness derives from changing seed

What does "sampling" over the model distribution mean in this context?
It simply means to provide an identical prompt to LLM several times changing the seed at each iteration. **Notice:** By def. , as soon as you change temperature, you change the model (and so you are sampling in a different model distribution!)

Take home...

An output (e.g. a job title) is a sequence of tokens. Its probability is the product of next-token probabilities at each step (chain rule).

If you change T, you change each step's next-token distribution, so you sample on a different model so the product changes.

Keep in mind: if you claim that you bootstrapped the LLM you cannot claim it by changing temperature at each iteration!

Final clarifications

Transformers = LLMs?

We now (hopefully) have clear what is a transformer and what is an LLM.

However their difference can be tricky to catch since transformers are necessary and sufficient conditions for having an LLM!

More generally however, a transformer is a complex NN with attention mechanism that can be fine-tuned and trained on every type of data (e.g. CAREER does it on numeric variables) and whose output is a probability distribution.

An LLM is a transformer specifically trained on language. Typically these are trained on language and while the output is a probabilistic distribution over words.

Output of a transformer: It is the output of a NN: probabilities.

Output of an LLM: Formally these are probabilities over the next word, however these are not seen by the user $\rightarrow$ unless you have constructed the LLM yourself, there is no easy way to see the full probability distribution over all the vocabulary (nowadays maybe yes) $\rightarrow$ the user can only see the next word(s)!

Fine tuning vs Prompting

Papers use these terms in a confusing way often when talking about LLMs.
True meaning:

- Fine-tuning in theory refers only to the fine-tuning of the parameters of the transformer inside the LLM: taking a pre-trained LLM and continuing its training on a smaller, specialized dataset to adapt its internal parameters for a specific task or domain $\rightarrow$ **Fine-Tuning changes permanently the parameters of the pre-trained LLM specializig it;**
- Prompting is the set of sentences that we write on the interface of a GPT to get an answer.

Confusing usage:

- Some papers (above all those written when LLMs started to emerge and we knew little about them) say that they "fine-tuned" the LLM in order to obtain a certain result meaning that they corrected the answers of the LLM after prompting.
- The practice to "teach" LLMs via prompting them documents, CVs...is in reality know as Prompt engineering $\rightarrow$ **Prompt engineering does NOT change the parameters of the pre-tained LLM, but just refines its answers**

The importance of parameters

While smaller model trained on high-quality data can outperform a larger model trained on poor data, **ceteris paribus** having more parameters allows to capture more complex relationships (i.e. to get closer to human language).

Which parameters? Parameters are spread throughout the LLM: e.g. embedding and positional weights, attention weights, all the parameters of the FFN, Top-p (number of tokens considered possible new predicted words),...

Why **ceteris paribus,** more parameters is better?

Think about the benefits of parameters: e.g. more FFN parameters means more refinement and more possibilities to explore non-linear relationships in input data and in output; more attention weights means a better understanding of the relationships of the words one to the other,...

Of course more parameters come with a cost: **more computational power needed!**

Temperature

Temperature control randomness in token selection: the higher the temperature the more diverse (also low-likely) outcomes are considered. Formally, temperature rescales logits before softmax:

$$p_T(i) = \frac{exp(z_i/T)}{\sum_j exp(z_j/T)}$$

where z_i and z_j are logits (outputs of decoder).
T=1 gives the model's own next-token distribution; T<1: high-logit options get more mass; T>1: flatter mass spreads to lower-logit options.

Why? Intuition in next slide...

Premise: Logits are the raw, unnormalized output scores generated by a model before they are converted into probabilities.

They represent the model's confidence in each possible output token or class. Higher logit values indicate higher confidence.

Look at the *odds ratio* between two tokens i and j:

$$\frac{p_T(i)}{p_T(j)} = \exp\left(\frac{z_i - z_j}{T}\right).$$

If $z_i > z_j$ (token i has a higher logit), then $(z_i - z_j)/T$ gets larger when $T < 1$. Hence the ratio $\frac{p_T(i)}{p_T(j)}$ increases, so the higher-logit token grabs *more* probability mass relative to the lower-logit one. Conversely, $T > 1$ shrinks that ratio (flatter distribution).

Labor-LLM (the paper)

Motivation

- **Why interesting for Labor economists?** Provide reliable predictions of the conditional probability of transitions in labor market;
- **Tons of "traditional models" failed. Why?** Tons of career paths for a person, but few data points of people that effectively undertook them! $\rightarrow$ need to predict the 0s somehow! traditional models make too restrictive assumptions (e.g. next occupation of a worker depends only on his/her last occupation) $\rightarrow$ inadequate for predictions
- Solution? **foundation models.** Models that are able to encode (transform in a low dimensional vector) the entire career history of a worker. Foundation models are basically the resulting models after pre-training (e.g. LLMs after pre-training on corpus).
- **Idea:** Predicting next job is similar to predicting next word which can be done via LLM!
- The power of foundation models lies on the fact that they are pre-trained using large-scale datasets (e.g. the internet) and lately fine-tuned using specialized surveys. Examples are transformers (and LLMs).

Challenges

Challenges with General (no FT) LLMs

- **General-purpose LLMs (i.e. not fine-tuned)** may not accurately represent real-world job transition probabilities (usually not trained on representative-data: no guarantee that the transition specified by LLM reflects the true one);
- Without fine-tuning, LLMs might not reflect true transition patterns for diverse demographics (e.g., underrepresented groups).

Challenges with FT-LLMs

FT.1 Even fine-tuned LLM present a challenge (res. question basically): can FT-LLMs make predictions about job transitions that are **representative** of real-world transitions, **conditional on history**?

FT.2 **Why challenging?** Because the true population transition probabilities are unknown due to the high dimensional space of potential histories!

FT.3 The idea is that –since I could have done tons of other jobs, and hence condition on tons of possible histories – the true (conditional) transition probabilities from my actual job to **any possible other jobs** are unknown!

FT.4 **How to evaluate performances then** since we don't know the true conditional probabilities of next occupations for every possible history (the "ground-truth transition law")? We don't need the true probabilities to **score predictions.** We only need the actual realized next job for each **observed** history!

Intuition: We want to make prediction and understand how good the model is in predicting next job! So as long as the model predicts accurately the actual job (next job for the fine-tuned model in the training set) and has outer-validity, who cares about having all possible histories!

FT.5 So we cannot report "true transition probabilities" for each complex history (a representativeness problem), but we can evaluate predictions on **observed** outcomes.

FT.6 **Disclaimer:** This assesses predictive accuracy, not the full, unknown data-generating law (e.g. **I cannot say with certainty what would have been your next job would the history be different** or in general the prob. over other possible next job given all possible histories).

FT.7 So... good for prediction not for counterfactuals.

Prompting or Fine-tuning?

In this paper they fine-tuned the model adapting pre-trained models to a specific domain: Labor.

So they properly fine-tuned the transformer updating the weights of a **pre-trained** (remember the pre-training on the corpus? They do not do it but take a pre-trained model for computational reasons) LLM using labor data (CV...) $\rightarrow$ as if we take someone who knows the general rules of the world and specialize her. Called **FT-LABOR-LLM**.

Embedding Extraction + Classifiers: Use LLM-generated embeddings with separate classifiers to refine predictions.

Robustness checks with prompt eng. and in context-learning made but lower performance. Remember? these do not require fine-tuning: you can also do it directly in chatgpt interface by making it more an more specific questions...

Objective

The paper claims that, by using fine-tuned LLMs (Llama-2) rather than a fine-tuned transformer architecture as in CAREER, there are major advantages.

Why?

The key differences are the following: (i) the transformer that forms CAREER has fewer parameters (a major disadvantage for a transformer as seen) (ii) CAREER is pre-trained on a much smaller corpus than Llama-2 (iii) CAREER does not use textual descriptions of job-titles $\rightarrow$ important to do so because we can then represent full career histories in lower dimensional embeddings...

LLM Notation

Notational conventions in LLM: The setup defines a word set $\mathcal{W}$, a token vocabulary (set of all possible tokens) $\mathcal{V}_{\text{LLM}}$ (e.g., size $32{,}000$), and a tokenization map $\text{TOK} : \bigcup_{j=1}^{\infty} \mathcal{W}^j \to \bigcup_{j=1}^{\infty} \mathcal{V}_{\text{LLM}}^j$ that converts any ($\bigcup_{j=1}^{\infty}$) word sequence into a token sequence. Given a context limit C_{LLM}, the LLM estimates next-token probabilities $\widehat{P}_{\text{LLM}}(v_{k+1} \mid v_{1:k}) \in [0, 1]$ for contexts $v_{1:k} \in \mathcal{V}_{\text{LLM}}^k$ with $0 \leq k \leq C_{\text{LLM}}$.

Notice: the "avalanche (chain) rule" mentioned at pag.55, can be translated in the current notation by saying that the sequence $\widehat{P}_{\text{LLM}}(v_{k+1}, v_{k+2} \ldots v_{k+k'} \mid v_{1:k})$ can be derived from individual next token predictions.

Classical objective (see above): estimate the probability that the next token is v_{k+1}, conditional on a sequence of k tokens (i.e. the prompt) using the FT-LLM.

Additional functions are as follows:

- $\text{TITLE}: \mathcal{Y} \longrightarrow \bigcup_{j=1}^{\infty} \mathcal{W}^j$ maps an occupation to its English-language title. For example, the occupation with `occ1990dd` code 95 is mapped to "nurse practitioners."
- $\text{TMPL}\left(x_{i,\leq t},\, y_{i,\leq T_i}\right)$ transforms the person's **full resume** into text all covariates up to time t and every observed occupation from the start through the last observed transition T_i. Used for fine-tuning.
- $\text{TMPL}\left(x_{i,\leq t},\, y_{i,<t}\right)$ The *truncated resume* up to (but not including) transition t contains covariates up to t and only past occupations $1, \ldots, t-1$. This is the prompt used at prediction time for transition $t \leq T_i$; the occupation at t is intentionally omitted so the model can predict it.

Labor Notation

1. Difference between $t = 1, 2, \ldots T_i$ = sequence number of observation events (transitions) for person i and $year_{i,t}$ representing calendar year at transition t.
2. Transitions $\neq$ change jobs! They are just checkpoints were they record the occupation of i: sometimes it changed sometimes it did not.
3. Why not calling t the time? Because the checkpoints are unevenly spaced in calendar years (i.e. $t+1$ for i can be 3 years after). The model predicts what happens from this checkpoint to the next one, regardless of whether that's 1 year or 3 years apart. Summary example:

event (t)	*calendar year* $(\text{year}_{i,t})$	*occupation* $(y_{i,t})$
1	2010	Sales
2	2012	Sales
3	2015	Tech

Here, t advances $1 \to 2 \to 3$ as observations occur, while years jump $2010 \to 2012 \to 2015$. The model predicts the state at $t+1$ given the history through t, regardless of how many calendar years pass between t and $t+1$.

3. $y_{i,t}$ occupation of i in transition index t;
   $y_{i,<t} = (y_{i,1}, y_{i,2}, \ldots y_{i,t-1})$ occupations of i prior to t with $y_{i,1} = \varnothing$ (i.e. unemployed in $t = 0$);
4. $x_{i,\leq t}$ are the time varying (support $\mathcal{X}_{var}$) and time invariant (support $\mathcal{X}_{inv}$) characteristics of i before and including t.
5. Probability that worker's next job is $y_{i,t}$ conditional on $i's$ prior career history $y_{i,<t}$ and (prior) covariates:

$$P(y_{i,t} \mid y_{i,<t}, x_{i,\leq t})$$

As already mentioned, they cannot know the DGP, say
$P^*(y_{i,t+1}|\text{full history up to } t) \quad \forall$ histories. However the can evaluate the performance of the model on observed next jobs $y_{i,t}$.

How so? Via **perplexity**: lower perplexity, more accurate performance. For an occupation model $\hat{P}(y_{i,t}|x_i, x_{i,\leq t}, y_{i,<t})$ (estimated probability via LLM that the LLM assigns to occupation y in t)

$$PPL = \exp\left(-\frac{1}{\sum_{i=1}^{N} T_i} \sum_{i=1}^{N} \sum_{t=1}^{T_i} \log P(y_{i,t} \mid x_i, x_{i,\leq t}, y_{i,<t})\right)$$

Logic of perplexity: exponential of the average negative log-likelihood (aka cross-entropy);
If $PPL = |\mathcal{Y}|$, the model behaves as if at each step it was choosing among $|\mathcal{Y}|$ equiprobable occupations (tiny link with discrete choice models).
If PPL $= 1$, the model is (essentially) perfectly confident and correct every time.

Aim is o minimize PPL!

How much would perplexity be if the model is uninformative (i.e. each next job equally likely?)

How much would perplexity be if the model is uninformative (i.e. each next job equally likely?) $|\mathcal{Y}|$, i.e. nr. of jobs considered.
Logic: fix a guy, i, $T_i = 5$ and nr. of jobs is $|\mathcal{Y}|$. The uninformative model will predict all next jobs as equally likely with prob.: $\frac{1}{|\mathcal{Y}|}$. The uninformative model assigns prob. $\frac{1}{|\mathcal{Y}|}$ to each next job. So we have
$PPL = exp(-\frac{1}{5} * 5 * (log(1/|\mathcal{Y}|))) = |\mathcal{Y}|$.

Where randomness can arise? 1) Which training people you happened to use (training-sample luck). 2) Randomness inside fine-tuning itself (e.g., the shuffled order of examples used by stochastic gradient descent). 3)Which test people you happened to evaluate on (test-sample luck).
What is point 2)? In fine-tuning, the model sees the training cases many times (epochs). Before each epoch, we shuffle the order of training examples (or mini-batches). SGD updates weights after each example/mini-batch using its gradient. So **randomness comes from shuffling in fie-tuning...**
Why shuffling at all? (i) Shuffling breaks harmful patterns (e.g., all nurses then all programmers) and (ii) makes gradient noise more "i.i.d.

Possible Solution: BOOTSTRAP

What we have until now?

- **LLM Notation:** Functions that transform numerical variables into texts and collection of variables (e.g. $y_{i,\leq t}$, $x_{i,\leq t}$) into text:
  - $\mathrm{TITLE} \rightarrow$ job codes into plain english;
  - $\mathrm{TMPL}\left(x_{i,\leq t}, y_{i,\leq T_i}\right) \rightarrow$ FULL resume into text;
  - $\mathrm{TMPL}\left(x_{i,\leq t}, y_{i,\leq t}\right) \rightarrow$ TRUNCATED resume (up to t) into text;

  useful for prompting and fine-tuning;
- **Labor Notation:** Definitions of career history, transition probabilities and the way to express covariates up to a certain point. $\rightarrow$ **useful for math clarity.**
- A way to measure the qualitiy of predictions in LLMs **(perplexity)**;

With these tools we can distinguish the 3 uses of LLMs in the paper better:

1) **Model used only as generative (no output probabilities):** Give LLM the truncated résumé prompt. The model writes a job title as plain text, e.g. it outputs: "Software Engineer." That's just a guess produced by its decoder. **Does not tell how likely other jobs were!**.

2) **Model outputs probabilities** (for a given model distribution, i.e. prompt, temperature...):

- Provide the truncated résumé as the *prompt history* (context).
- For a candidate job title y with tokenization $(t_1, \ldots, t_n)$, query the LLM for next-token probabilities (provided by model this time) at each step.
- Score y via the chain rule:

$$\widehat{P}_{\mathrm{LLM}}(y \mid \text{history}) \;=\; \prod_{j=1}^{n} \widehat{P}^{V}_{\mathrm{LLM}}\big(t_j \;\big|\; \text{history}, t_{<j}\big).$$

Example follows in next slide

- (Optional, only if a full distribution over a finite candidate set Y is desired.) Evaluate the product above for *all* $y \in Y$ and normalize:
- Note: These are the model's own probabilities for the given prompt and settings.

- Tokens for $y_1 =$ "Software Engineer": $t_1 =$ Software, $t_2 =$ Engineer, then end-of-title $\langle \mathrm{EOT} \rangle$.
- Tokens for $y_2 =$ "Nurse": $t_1 =$ Nurse, then $\langle \mathrm{EOT} \rangle$.

Model stepwise probabilities (given the history).

$$\begin{aligned} P(\text{Software} \mid \text{history}) &= 0.50, \\ P(\text{Engineer} \mid \text{history, Software}) &= 0.60, \\ P(\langle \mathrm{EOT} \rangle \mid \text{history, Software Engineer}) &= 0.80; \end{aligned}$$

$$\begin{aligned} P(\text{Nurse} \mid \text{history}) &= 0.40, \\ P(\langle \mathrm{EOT} \rangle \mid \text{history, Nurse}) &= 0.90. \end{aligned}$$

Chain-rule scores (model probabilities for full titles).

$$\widehat{P}_{\mathrm{LLM}}\big(\text{"Software Engineer"} \mid \text{history}\big) = 0.50 \times 0.60 \times 0.80 = \mathbf{0.24}.$$

$$\widehat{P}_{\mathrm{LLM}}\big(\text{"Nurse"} \mid \text{history}\big) = 0.40 \times 0.90 = \mathbf{0.36}.$$

So we have a probability for each job title.

3) **Use Embeddings directly:** Some LLMs expose a lower-dimensional *embedding function* that maps any token sequence to a real vector (e.g. for Llama-2-7B LLM is $d_{\mathrm{LLM}} = 4096$). Formally the embedding function (transforms tokens in vectors of dim. 4096) is

$$\mathcal{E}_{\mathrm{LLM}} : \bigcup_{j \le C_{\mathrm{LLM}}} \mathcal{V}_{\mathrm{LLM}}^{j} \longrightarrow \mathbb{R}^{d_{\mathrm{LLM}}},$$

Given a word string s, tokenization $\mathrm{TOK}(s)$ produces a sequence $(v_1, \ldots, v_k) \in \mathcal{V}_{\mathrm{LLM}}^{k}$, and the composite map $\mathcal{E}_{\mathrm{LLM}} \circ \mathrm{TOK}$ returns its embedding:

$$e(s) \;=\; \mathcal{E}_{\mathrm{LLM}}\big(\mathrm{TOK}(s)\big) \in \mathbb{R}^{d_{\mathrm{LLM}}}.$$

Mock example. Let $s_1 =$ "Past jobs: cashier $\rightarrow$ barista" and $s_2 =$ "Past jobs: teller $\rightarrow$ barista". Compute

$$e_1 = \mathcal{E}_{\mathrm{LLM}}\big(\mathrm{TOK}(s_1)\big), \qquad e_2 = \mathcal{E}_{\mathrm{LLM}}\big(\mathrm{TOK}(s_2)\big) \in \mathbb{R}^{4096}.$$

A common use is similarity via cosine: $\cos(e_1, e_2) \;=\; \frac{e_1^{\top} e_2}{\|e_1\| \, \|e_2\|} \in [-1, 1]$, which measures how close the two histories are in the LLM's representation space.

Most commonly they are **used in tandem with MNL as covariates ($z_{i,t}$ in later slides).**

Step 1) vs Step 2) vs Step 3).

- **LLM as text generator only.** In Step 1 they **DECIDE** to use LLM as generative model: no probabilities/logprobs are read (so no chain rule can be applied). **Could we recover probs?** One could in principle **make sampling** (see pag. 57) on the (model) distribution with temperature $\hat{T}$? Yes but is (a) costly + (b) rare-event issue! (c) useless for robustness: same functionality as normal LLM so why not using it directly?...
  - (a): To obtain low-variance: Many samples per temperature + Many temperatures;
  - (b): **Pick the same sequence of tokens exactly at every sample is difficult!** Hence variance is hard to measure. Words are sequence of tokens! **Stupid example:** "Software Engineer" is several tokens. Its probability is the product of stepwise next-token probabilities. Products get small fast $\rightarrow$ the exact string is often a rare event over multiple sample (e.g. sometimes it can be "Software analyst");
- Step 2 (logprobs): **LLM provides logprobs** and apply the chain-rule scoring of any candidate job.
- Step 3 (embeddings): treat the LLM as a **feature extractor** $\rightarrow$ map any text (e.g., résumé history or a job title) to a fixed-length vector $\rightarrow$ compare vectors (or use NN/MNL for predictions)

Benchmark models

The model with which results for CAREER and LABOR-LLM are confronted with **benchmark models**.

First bench. comes from economic theory and is NOT an occupational model: **Multilogit (MNL)**!

Idea: we have a **choice problem** among $\mathcal{Y}$ occupations. For person i at time t, build a feature vector $z_{i,t} = g(x_{i,\leq t}, y_{i,<t})$ (history $\rightarrow$ covariates).
NOTICE: $z_{i,t}$ **can be *hand-crafted indicators* (pure benchmark) or *LLM embeddings* of the career history (see Step-3) (LLM complement)**

Each alternative $y \in \mathcal{Y}$ has a parameter vector β_y. The latent utility is

$$U_{i,t}(y) = z_{i,t}^{\top}\beta_y + \varepsilon_{i,t}(y),$$

with $\varepsilon_{i,t}(y)$ i.i.d. Gumbel.
This yields the *softmax* choice probabilities

$$\widehat{P}_{\mathrm{MNL}}(y_{i,t} = y \mid x_{i,\leq t}, y_{i,<t}) = \frac{\exp\left(z_{i,t}^{\top}\beta_y\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(z_{i,t}^{\top}\beta_{y'}\right)}.$$

Parameters $\{\beta_y\}_{y \in \mathcal{Y}}$ are estimated by (regularized) maximum likelihood.

Second benchmark model is the **empirical model**.
Is a model that does not use any covariates or other information beyond the immediately preceding occupation to make predictions:

$$\widehat{P}_{\text{Empirical}}(y_{i,t} \mid x_{i,\leq t},\, y_{i,<t}) = \frac{\#^{(\text{train})}\{\, y_{i,t-1} \to y_{i,t} \,\} + 1}{\#^{(\text{train})}\{\, y_{i,t-1} \,\} + 1}\,.$$

$\to \#^{(\text{train})}\{\, y_{i,t-1} \,\} + 1$: number of times occupation y appears in the training data in $t-1$;
$\to \#^{(\text{train})}\{\, y_{i,t-1} \to y_{i,t} \,\}$: number of times the transition from occupation $y_{i,t-1}$ to $y_{i,t}$ appears in the training data;
$\to$ "+1" added for avoid dividing by 0.

CAREER

Big picture: CAREER adopts MNL with feature vectors $z_{i,t}$ learned via Transformer to predict the next occupation in two stages: (1) will the person stay or switch? (2) if switch, to which occupation.

Pre-requisites CAREER learns an embedding function $\mathcal{E}_{\text{CAREER}}$ that maps history and covariates to a fixed-length vector;
The output at time t is written

$$h_{i,t}^{(L)}\big(x_{i,\le t},\, y_{i,<t}\big) \in \mathbb{R}^{d_{\text{CAREER}}},$$

the *final-layer L representation* summarizing the person up to t.

The first embedding layer combines the latest job and covariates:

$$h_{i,t}^{(1)} \;\equiv\; e_{\text{occupation}}(y_{i,t-1}) \;+\; e_{\text{static}}(x_i) \;+\; e_{\text{dynamic}}(x_{i,t}) \;+\; e_{\text{time}}(t)\,.$$

Each $e(\cdot)$ is a learned embedding; later layers refine this via transformer blocks (self-attention + FFN) as follows:
Across layers $\ell = 1, \ldots, L$, CAREER aggregates past representations using attention weights $\pi_{i,t,t'}^{(\ell)}$ over previous times $t' \le t$:

$$\tilde{h}_{i,t}^{(\ell)} \;=\; h_{i,t}^{(\ell)} \;+\; \sum_{t'=1}^{t} \pi_{i,t,t'}^{(\ell)} \;*\; h_{i,t'}^{(\ell)}, \qquad h_{i,t}^{(\ell+1)} \;=\; \mathrm{FFN}^{(\ell)}\Big(\tilde{h}_{i,t}^{(\ell)}\Big). \tag{1}$$

This yields the final summary vector $h_{i,t}^{(L)}$.

Toy attention update (one worker) not in paper

%%% EXAMPLE START

At checkpoints $t = 1, 2, 3$ we have

$$h_{i,1}^{(\ell)} = (1, 0), \quad h_{i,2}^{(\ell)} = (0, 1), \quad h_{i,3}^{(\ell)} = (0.5, 0.5).$$

At $t = 3$, for instance the layer asks: *"How much should I borrow from $t = 1$ (e.g., nursing license), from $t = 2$ (recent hospital role), and from $t = 3$ (current status) to form the best summary for making the $t \to t + 1$ prediction?"* The learned attention weights $\pi_{i,3,1}^{(\ell)}, \pi_{i,3,2}^{(\ell)}$, answer that:

$$\pi_{i,3,1}^{(\ell)} = 0.2, \qquad \pi_{i,3,2}^{(\ell)} = 0.7.$$

The above weights are attention scores learned by gradient descent as at pag. 21 via backpropagation.

Namely, using the above notation... In layer ℓ, queries Q, keys K, and values V are obtained via learned projections:

$$Q_{i,t} = h_{i,t}^{(\ell)} W_Q^{(\ell)}, \qquad K_{i,t'} = h_{i,t'}^{(\ell)} W_K^{(\ell)}, \qquad V_{i,t'} = h_{i,t'}^{(\ell)} W_V^{(\ell)}.$$

Why Q,K,V functions of $h_{i,t}^{(l)}$?...See next slide for deepening...

As seen from an embedding vector ($h_{i,t}^{(l)}$ in our case) we make three role-specific views represented by Q, K, V which are "different divisions of the space where $h_{i,t}^{(l)}$ lies", i.e. these views are produced by learned linear projections so the model can discover which subspace of h is useful for matching (Q · K) and which for content (V) via weights W_Q, W_K, W_L.

With a causal mask ($t' \leq t$), the attention weights are *computed* as

$$\pi_{i,t,t'}^{(\ell)} \;=\; \mathrm{softmax}_{t' \leq t}\left(\frac{Q_{i,t} K_{i,t'}^{\top}}{\sqrt{d_k}}\right),$$

and the context-mixed vector is

$$\tilde{h}_{i,t}^{(\ell)} = h_{i,t}^{(\ell)} + \sum_{t' \leq t} \pi_{i,t,t'}^{(\ell)}\, V_{i,t'}, \qquad h_{i,t}^{(\ell+1)} = \mathrm{FFN}^{(\ell)}\big(\tilde{h}_{i,t}^{(\ell)}\big).$$

Key point: the matrices $W_Q^{(\ell)}, W_K^{(\ell)}, W_V^{(\ell)}$ and the FFN parameters are the ***learned*** quantities (via backprop from the loss) $\rightarrow$ once learned we can get the attention weights $\pi_{i,t,t'}^{(\ell)}$.
Notice: when backprop. is over the whole FFN (i.e. for all layers) has optimal weights. So we can talk about "optimal $\pi_{i,t,t'}^{(\ell)}$" (i.e. attention scores computed with optimal parameters in layer ℓ) for instance.

Given that backprop. is over and we have optimal values for a layer ℓ...the context-augmented vector and layer update:

$$\tilde{h}_{i,3}^{(\ell)} = h_{i,3}^{(\ell)} + 0.2\, h_{i,1}^{(\ell)} + 0.7\, h_{i,2}^{(\ell)} = (0.7, 1.2), \qquad h_{i,3}^{(\ell+1)} = \mathrm{FFN}^{(\ell)}\big(\tilde{h}_{i,3}^{(\ell)}\big).$$

Summary. The transformer layer forms a new summary for time t by mixing the *current* representation with a *weighted average* of all past checkpoints (weights π learned from data), then applies a nonlinear FFN. This lets the model emphasize recent or historically important states before the two-stage head predicts (i) move vs. stay and (ii) if moving, the destination.

Notice: The above example must be read as either (i) a snapshot during training using the current parameter set at step k of optimization, or (ii) as post-training using the fitted parameter set.
This is because we plugged in fixed numbers for the attention weights (e.g., 0.2, 0.7) and then just compute $\tilde{h}$ and pass it through the trained FFN.

Why CAREER use last layer L h? Because by the time you reach layer L, the representation at time t has already integrated **the whole usable history (via the causal self-attention mixes in all $t' \le t$ at every layer)**, and training has shaped that very layer to be the thing the heads can read most easily.
%%% EXAMPLE END

CAREER is a 2-stage algorithm...
First Stage. *How likely is a change of occupation at $t+1$?* Logistic (sigmoid) function of the final embedding:

$$\widehat{P}_{\text{CAREER}}(\text{move}_{i,t} = 1 \,|\, x_{i,\leq t},\, y_{i,<t}) = \frac{1}{1 + \exp\Big(- \eta^\top h_{i,t}^{(L)}(x_{i,\leq t},\, y_{i,<t})\Big)}.$$

Why this form? Choice model: switch to other job vs outside option (MNL with outside opt.).
Second Stage. Let $\beta \in \mathbb{R}^{d_{\text{CAREER}} \times |\mathcal{Y}|}$ collect one weight vector β_y per occupation y. If $\text{move} = 1$ choice model between careers (MNL):

$$\widehat{P}_{\text{CAREER}}(y_{i,t} = y \,|\, x_{i,\leq t},\, y_{i,<t},\, \text{move}_{i,t} = 1) = \frac{\exp\big\{\beta_y^\top h_{i,t}^{(L)}(x_{i,\leq t},\, y_{i,<t})\big\}}{\sum\limits_{y' \neq y_{i,t-1}} \exp\big\{\beta_{y'}^\top h_{i,t}^{(L)}(x_{i,\leq t},\, y_{i,<t})\big\}}.$$

If no move (MNL with $y = y_{i,t-1}$),
$\widehat{P}_{\text{CAREER}}(y \,|\, x_{i,\leq t},\, y_{i,<t},\, \text{move}_{i,t} = 0) = \mathbf{1}\{y = y_{i,t-1}\}$.
Combine Stage 1 and 2:

$$\widehat{P}_{\text{CAREER}}(y \,|\, x_{i,\leq t},\, y_{i,<t}) = \begin{cases} 1 - \widehat{P}_{\text{CAREER}}\Big(\text{move}_{i,t} = 1 \mid \cdot\Big), & \text{if } y = y_{i,t-1} \\ \widehat{P}_{\text{CAREER}}\Big(\text{move}_{i,t} = 1 \mid \cdot\Big)\ \widehat{P}_{\text{CAREER}}\Big(y \mid \cdot,\, \text{move}_{i,t} = 1\Big), & \text{if } y \neq y_{i,t-1} \end{cases}$$

COMPARING PERFORMANCE OF OCCUPATION MODELS

Thus the models being compared are:

- $\widehat{P}_{\text{LLM}}(y \mid \text{history})$ — end-to-end next-occupation prediction from the raw history using a language model. Used in 3 ways (see from pag. 80):
  - Generative only
  - With output probabilities
  - Embeddings directly (used with MNL for making the prediction)
- $\widehat{P}_{\text{CAREER}}(y \mid x_{i,\leq t}, y_{i,<t})$ — two-stage MNL (stay/switch then destination) using a transformer summary $h^{(L)}$ of the history.
- $\widehat{P}_{\text{MNL}}(y_{i,t} = y \mid x_{i,\leq t}, y_{i,<t})$ — conventional MNL on hand-crafted covariates (no learned embeddings/attention).
- $\widehat{P}_{\text{Empirical}}(y_{i,t} \mid x_{i,\leq t}, y_{i,<t})$ — baseline choice model measuring the probability of transitioning given as only covariate the last occupation $y_{i,t-1}$.

$\widehat{P}_{\text{MNL}}(y_{i,t} = y \mid x_{i,\leq t}, y_{i,<t})$ vs $\widehat{P}_{\text{CAREER}}(y \mid x_{i,\leq t}, y_{i,<t})$

$\widehat{P}_{\text{MNL}}(y_{i,t} = y \mid x_{i,\leq t}, y_{i,<t})$ with embeddings of career history created via LLM (i.e. $z_{i,t}$ is created using $\mathcal{E}_{\text{LLM}}$). Hence LLM is only used here as a feature extractor (Step-3 of pag. 80), i.e. no passing through encoder-decoder architecture.

vs

$\widehat{P}_{\text{CAREER}}(y \mid x_{i,\leq t}, y_{i,<t})$ (explained above)

Winner (performance based on perplexity): CAREER even if it relies on less info! (does not utilize birth year information).

Any explanation? 1) $\widehat{P}_{\text{MNL}}(y_{i,t} = y \mid x_{i,\leq t}, y_{i,<t})$ uses an embedding of career history without self-attention, which instead is exploited in CAREER; 2) The LLM provides an embedding without being fine-tuned on that task (the LLM is only a feature extractor here): Only the MNL weights are fit! In CAREER instead embedding is learned end-to-end for next-occupation prediction.

So why using an LLM at all for embeddings? a) LLM is highly pre-trained on a huge corpus (knows what is an "industry", a "job" ecc...for free!) b) Provides an embedding already knowing that it is used for some prediction

	CAREER	
vs	***Win***	***Lose***
LLM embeddings + MNL	✔	

OTS $\widehat{P}_{\text{LLM}}(y \mid \text{history})$ vs $\widehat{P}_{\text{CAREER}}(y \mid x_{i,\leq t}, y_{i,<t})$

Off-the-Shelf (OTS) $\widehat{P}_{\text{LLM}}(y \mid \text{history})$ refers to applying Step-2 of pag. 80 on a not fine-tuned nor prompted-eng. LLM (i.e. as if you take the model of GPT4, instead of prompts you input embeddings at the give temperature...and take the output probabilities $\widehat{P}_{\text{LLM}}(y_{i,t} \mid x_{i,\leq t},\, y_{i,<t}) \overset{\text{def}}{=} \widehat{P}^{\mathcal{V}}_{\text{LLM}}\big(\text{Tok}\big(\text{Title}(y_{i,t})\big) \,\big|\, \text{Tok}\big(\text{Tmpl}(x_{i,\leq t},\, y_{i,<t})\big)\big)$.)

VS

$\widehat{P}_{\text{CAREER}}(y \mid x_{i,\leq t}, y_{i,<t})$ (explained above)

Winner (performance based on perplexity): CAREER

Any explanation?

- **Not adapted to our data:** Off-the-shelf LLMs are pretrained on generic text, not on these survey career sequences – so they miss the survey's priors, i.e. the descriptives that can be learned (e.g., high "stay" rate, common/rare transitions, cohort patterns).
- **Unknown label set:** The task's outputs are a fixed list of valid occupations (OCC1990dd + specials). Generic LLMs spread probability over *any* string, including invalid job titles, which dilutes accuracy unless constrained or fine-tuned.

	CAREER	
vs	***Win***	***Lose***
LLM embeddings + MNL	✔	
Off-the shelf LLM	✔	

PE $\widehat{P}_{\text{LLM}}(y \mid \text{history})$ vs $\widehat{P}_{\text{CAREER}}(y \mid x_{i,\leq t}, y_{i,<t})$

Prompt Engineered (PE) LLMs, refer to OTS LLM improved by adding additional information into the prompt: 1) prepend the list of all 335 job titles (teaches the context where we are: we are looking for predictions in the space of these 335 job titles); 2) prepend CVs from other workers: informs OTS LLM about the structure of the data.

VS

$\widehat{P}_{\text{CAREER}}(y \mid x_{i,\leq t}, y_{i,<t})$ (explained above)

Winner (performance based on perplexity): CAREER

Any explanation? Reduced prompt length (in early GPTs): cannot pre-pend a lot of info so that OTS is not that much bettered...

	CAREER	
vs	***Win***	***Lose***
LLM embeddings + MNL	✔	
Off-the shelf LLM	✔	
Improved LLM via prompt eng.	✔	

FT $\widehat{P}_{\text{LLM}}(y \mid \text{history})$ vs $\widehat{P}_{\text{CAREER}}(y \mid x_{i,\leq t}, y_{i,<t})$

Fine-Tuned (FT) LABOR-LLM are LLMs fine tuned as explained at pages63 and 71.

Let's go deeper...: The fine-tuning takes several steps: 1) $\forall$ individuals i build text representation of their entire career ($TMPL(x_{i\leq T_i}, y_{i,\leq T_i})$) 2) Fine-tune the pre-trained LLM as follows: (i) since LLMs predict one token at a time, instead of training them to pick a job label directly, fine-tune the LLM as a language model on the templated career paragraphs; (ii) Once LLM knows the structure of career summaries, it can place very high probability on the specific tokens that spell valid job titles given the preceding context (history, year, etc.).

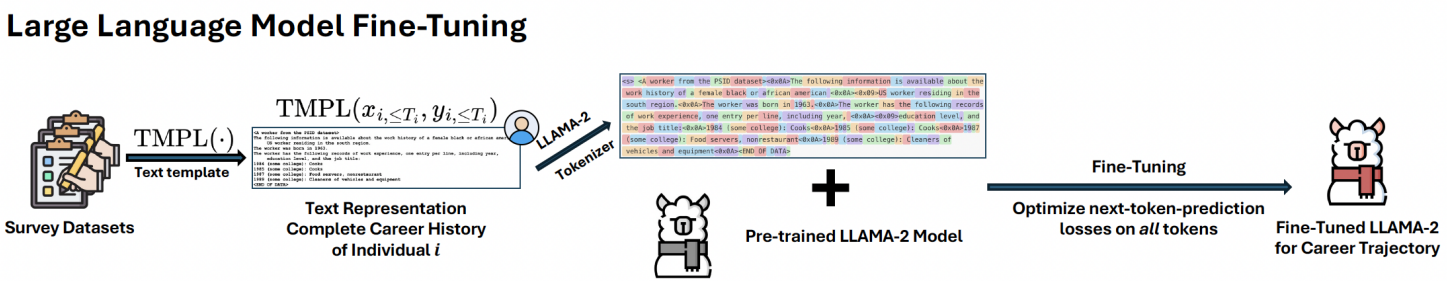


FIGURE 4. Illustration of the model fine-tuning procedure.

VS

$\widehat{P}_{\text{CAREER}}(y \mid x_{i,\leq t}, y_{i,<t})$ (explained above)

Winner (performance based on perplexity): FT LABOR-LLM

Any explanation? See pag. 72. Important: without FT, CAREER outperforms LLM. Fine-tuning essential to remove hallucinations! Notice that CAREER uses an already "fine-tuned" transformer so the comparison is fair (fine-tuning slightly differs though since CAREER accepts only numeric inputs)!

	CAREER	
vs	***Win***	***Lose***
LLM embeddings + MNL	✔	
Off-the shelf LLM	✔	
Improved LLM via prompt eng.		✔
Improved LLM via fine-tuning		✔

Table 2: Quick summary of performances (perplexity) of CAREER vs other models (Win-Lose refers to CAREER)

Notice: Standard Errors (SE) are computed via bootstrap, exploiting the sample variation in the training set (resample which individuals/transitions land in the training split (bootstrap))

Federico's concern: Notice however that, as explained at pag. 57, another source of randomness is possible even keeping training sample fixed! A further correction might be needed to compute SE!

Other checks + conclusions

This paper explores a lot of other refinements, robustness checks (e.g. performance improvements when increasing/decreasing nr of parameters of LLM...) and alternative sampling and bootstrap strategies which go beyond the scope of the presentation.

The aim of the above slides was just to introduce the reader to the usage of LLM in Economics fo predictive tasks providing Athey et al. (2024) as an example.

I strongly recommend the interested reader to go through LMR (2025) for a general discussion on the use of LLMs in Economics.