Natural Language Processing (Fall 2025)

Taught by Professor John Hewitt, Columbia University

Notes by Benny Attar[*]

1 Introduction to NLP

We begin the semester by asking the question: what is language modeling? We generally can define it by two distinct questions:

1. How do we build computer programs that learn from human text?
2. How do we build computer programs that can understand *and* generate human text?

1.1 How does one represent text?

We begin with a base model of representing text:

```
    Columbia is located in Manhattan.
```

By observing the prefix of test, and what is filled in, we learn quite a bit about the world and the mechanics of that text. For example:

```
    Columbia University is in NYC.
    Columbia University is in Manhattan.
```

This is a string of unicode / ASCII characters. This is good for comparisons such as:

```
    Columbia University is in NYC != Columbia University is in Manhattan.
```

With this model, we can also ask which of two pairs of strings is more similar:

```
sim(Columbia is in Manhattan, Columbia is in NYC)
sim(Columbia is in Manhattan, Columbia is in the Ivy League)
```

The problem is however, that string edit distance does not correspond to intuitive similarity. While NYC and Manhattan are similar, it is equally as grammatically correct to say in Columbia is in the Ivy League. Which leads us to the question: how do we represent words?

[*]I am thankful to Professor Hewitt for his teaching this semester. These are my compliation of notes from lectures - all errors here are my own.

1.2 How does one represent words?

We assume we have a finite vocabulary and each word has an `int` id:

- Manhattan: 0
- NYC: 1
- Ivy League: 2

We represent this via "one-hot vector encoding", a vector the size of our vocabulary with a 1 at the correct index for the word and zeros everywhere else. When a word is represented as a vector, we call the vector a **word embedding**.

- Manhattan: $[1, 0, ..., 0]$
- NYC: $[0, 1, ..., 0]$
- Ivy League: $[0, ..., 1, ..., 0]$

And we can easily compute comparisons via the L_1 norm distance.

$$||v_{Manhattan} - v_{NYC}||_1 = [1 - 0] + [0 - 1] + [0 - 0] + ... + [0 - 0] = 2$$

However, since they are all one hot vectors, the distance for all of the vectors is the same! We need some representation that is strong *regardless* of character. We need to create word embeddings with *meaning*. One potential idea is to craft components of meaning, or properties of words, and then embed each word in the vocabulary with the value in that component. For example:

- A physical place
- A city
- An athletic conference

So then:

- Manhattan: $[1, 0, ..., 0]$
- NYC: $[1, 1, ..., 0]$
- Ivy League: $[0, 0, 1, ..., 0]$

And now when we compute the distances:

$$\text{D(Manhattan, NYC)} = 1$$

$$\text{D(Manhattan, Ivy League)} = 2$$

$$\text{D(NYC, Ivy League)} = 2$$

So by breaking similarities into categories we get stronger comparisons. What are some of the problems in this model?

- You have to exhaustively enumerate every potential distinction in word meaning.
- Words can come up in different contexts + double meanings. Ex: bank vs bank, their vs they're vs there.
- Bias
- It would be computationally impossible to enumerate every potential meaning of every potential word in every language.

As we'll see, *representing* text and *predicting* text will be each others solutions to this dilemma. Modern language models do not discard embeddings entirely, but instead treat them as inputs to a learned contextualization function.[1] We go over Transformers later in the course.

1.3 Representing and generating text jointly

We want to learn the properties of words through the process of predictions. This is **language modeling.**

```
Columbia is in
```

Knowing whether to predict either "Manhattan", "NYC", "in the Ivy League" depends on the context of the sentence and how often each word comes up based on what came before. Our prediction for the next word should be probabilistic.

Language modeling is the task of determining the probability distribution over sequences of words.

Let V be a finite vocabulary and $w \in V$ a word in V. A string (an ordered sequence of words) is $(w_1, ..., w_k) \in V*$ where $V*$ is finite combinations of the finite vocab. A language model defines a probability distribution over all such sequences as:

$$P(w_1, ..., w_k)$$

Where P is the likelihood of that exact sentence $w_1, w_2, \ldots$ occurring as a single coherent sentence. But since words are not independent, the probability of a word depends on the word before it. By the chain rule of probability:

$$P(w_1, w_2..., w_k) = \sum_{i=1}^{k} p(w_i | w_{<i})$$

Where $p(w_i|w_{<i})$ is the probability of the next word i given all the previous words before i. So for example we can write:

[1]Models such as BERT, DistilBERT, and Sentence-Transformers should be understood not as alternatives to this framework, but as concrete instantiations of it. In these models, static token embeddings are used only as inputs; the primary representations are the contextualized hidden states produced by a Transformer. Sentence-Transformer models further apply pooling and contrastive objectives to produce sentence-level representations. See here for an overview on vector databases which largely power this.

- P(Manhattan — Columbia is in) = .6
- P(NYC — Columbia is in) = .4
- P(the — Columbia is in) = .8
- P(the greatest city in the world — Columbia is in) = .0001
- ...

These probabilities form a distribution over the vocabulary, meaning:

$$\sum_{w \in V} P(w|\text{Columbia is in}) = 1$$

The model's prediction is the word with the highest conditional probability. As we'll see, reaching that distribution is what matters.

1.4 Creating a language model

Given that we learned that defining each word and creating word embeddings based on meaning is virtually impossible, we can can create simpler word representations. To do so, we randomly sample a vector in some dimensionality d and use that as a representation of a word:

1. $v_{Manhattan} =\sim U[-0.001, 0.001]^d = [-.0003, .004, ...]^\top$
2. $v_{NYC} =\sim U[-0.001, 0.001]^d = [-.00006, .001, ...]^\top$
3. $v_{IvyLeague} =\sim U[-0.001, 0.001]^d = [-.00036, .0011, ...]^\top$

With these representations, we can come to a much simpler prefix (or "context") where we just average all of the word embeddings we have so far.

$$h(w_{<i})\frac{1}{i}\sum_{i=1}^{i-1} v_{wj}$$

Why do we average?

1. Simplicity
2. Every word equally contributes
3. Cheap

While these three reasons should be (and will be) picked on and dissected, we accept this simple model at face value for now. For example, averaging does not distinguish between different orderings of the same string. In this case:

```
Dog bites man == man bites dog.
```

But alas, we leave the "context model" as it is for now.

Now that we have a representation of the prefix and context, we can form a probability distribution of the next word! The **softmax function** averages together all of the word embeddings for the prefix and predicts the next word by comparing that average to the embedding of each word in the vocabulary. Each word will have a probabilistic "score" with a likelihood of coming up. The more similar each words embedding is to the average so far, the higher the probability will be:

$$P(w_i|w_{<i}) = \frac{\exp(v_{w_i}^\top h(w_{<i})}{\sum_{j \in V} \exp(v_{w_i}^\top h(w_{<i})}$$

Where:

- v_{w_i} is the candidate next word.
- $h(w_{<i})$ is the vector average of all previous words.

So we take the dot product of the candidate next word and vector average of previous words. If they are similar, the value will be large. We exponentiate so the value will be positive and the more similar they will be the value will be larger.

Then, we divide to normalize by the sum of all the words in the vocabulary. Effectively, our formula is such:

$$P(word) = \frac{\text{candidate word score}}{\text{sum of all words score}}$$

This is a probability distribution because it sums to one and all terms of non-negative because of the exponentiation. The words compete against eachother to get a higher score and to be the next word sent to the output!

1.5 Improving our predictions

How do we make our predictions better?

This is where the language model will be *learning* from the text and updating its probabilities. The general process is as follows:

- Specify a summary of how bad our predictions are $\rightarrow$ loss function
- Tweak the representations in our model to slightly less badly $\rightarrow$ reduce the loss
- Repeat $\rightarrow$ continuous gradient descent

We are trying to make our predictions better by minimizing our loss function through tweaking the word embeddings. To do so, we create the expected values of our data distribution:

$$\mathbb{E}_{(w_1,\ldots,w_k)} \sim D[P(w_1, ..., w_k)]$$

Where D is any massive dataset (such as all documents on the internet). To make the probabilities, we use the logarithm as a monotonic function and minimize for convention.

$$\text{Minimize}\mathbb{E}_{(w_1,...,w_k)} \sim D[-\log P(w_1, ..., w_k)]$$

And we are minimizing this over the word embeddings!

$$\min_{v_{w:w\in v}} \mathbb{E}_{(w_1,...,w_k)} \sim D[-\log P(w_1, ..., w_k)]$$

So we are constantly minimizing our loss function over the word embeddings we have randomly created.

To minimize and tweak the word embeddings to make them better for predictions we use **gradient learning**. A gradient tells us the best way to (locally, slightly) tweak a continuous value in order to minimize a function value. The process is:

$$v_w = v_w - \epsilon * \nabla_{v_w} L$$

Where:

- v_w is the embedded vector for w.
- ϵ is the step size
- $\nabla_{v_w} L$ is the gradient pairing in direction to decrease the loss.

If the next word is wrong, we nudge by the gradient. If the next word is right, we don't move much. This way, after each output, the model *continuously improves* on its probabilities.

1.6 Summary

While modern LLMs do much more powerful things than just compute the similarities between words, the general framework we've build is standard.

1. Build a representation of a large body of text using **word embeddings**.
2. Create probability distribution for that vector representation using the **softmax function**.
3. Constantly improve probability distribution using **gradient-based learning**.

We are left with a few guiding questions:

1. How do we efficiently and meaningfully create a representation of the prefix?
2. How do we determine what our (finite) vocabulary will be?
3. How can the model improve on its predictions?

2 Tokenization

We begin by answering the first question we were left off with: how do we turn large bodies of texts and strings to discrete elements to make models work? And further, how do we determine our finite vocabulary?

To answer this, we define tokenization: mappings from strings of text to *discrete tokens* from our finite vocabulary (and back).

We begin by taking in a whole text of a file and enumerating its entire vocabulary:

```
Carl swims in the river bank and works at the bank.
```

We encode the sentence as a sequence of integers corresponding to the index of the word in the finite vocabulary.

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

So our entire finite vocabulary has 9 words (of course this would be done at a much larger scale). Although immediately, we see some issues:

1. The two instances of bank are different words with the same string.
2. We did not define our whitespace.
3. What do we do with the period at the end?
4. What happens if we have multiple space characters in a row?
5. **What if try to encode a new file using our vocabulary but a string of characters is not in our vocabulary?**

That last one is the most prescient. Say we want to encode:

```
Carl works at the local bank.
```

Like before, we have some words that are in our vocabulary. We can index them as before.

```
[1, 8, 9, 4, ?????, 6]
```

But local is not an element of V! Even in a vocabulary of potentially millions, this problem will always exist. There will *always* be strings of characters that we missed after we create our fixed V.

2.1 Possible solutions for the finite vocabulary problem

We call this problem the **out-of-vocabulary** problem. There are several solutions, each with their pros and cons.

2.1.1 Character Tokenization

We could map every string of characters that comes up that's not in V to some special "unknown" word. So our example above would be:

```
Carl works at the local bank.
[1, 8, 9, 4, UNK, 6]
```

This is great because it prevents all unknown words and has high coverage. But, it removes the meaning of potentially a lot of tokens and is highly inefficient.

2.1.2 Byte Tokenization

Isn't our vocabulary just a combination of really small units that compose into larger ones that then create strings? Every string is effectively just a sequence of bytes! So why don't we create a vocabulary just of bytes? While this does solve for the problem of preventing unknown words, each element in the vocabulary will have no meaning. Also, this will be highly inefficient across huge datasets.

2.1.3 The Subword Solution

So we need a middle ground between:

- Words $\rightarrow$ have meaning, but high loss.
- Bytes $\rightarrow$ lossless, but meaningless.

The key design principle is to have each element in the vocabulary to correspond to a (small) chunk of text, that when combined with other (small) chunks will form (larger) words with meaning. We estimate these small chunks from data, so that the common large chunks will get their own place in the vocabulary as well. Thus, we have high coverage (after combining chunks) and high level of meaning.

For example:

```
I went to the beeeeeaaach.
[I, _went, _to, _the, _bee, ee, a, aa, ch]
```

Every word that is preceded by a space has a preceding underscore. The `_bee` token will be different than a potential `bee` token that can come up elsewhere. Next, the rare words, `ee`, `aa` and `ach`, are split into multiple pieces because they can be elements of other strings (such as `aardvark`).

The intuition of subword vocabularies is that smaller uncommon strings will be building blocks for larger strings. If bytes often co-occur, they should show up in a token together.

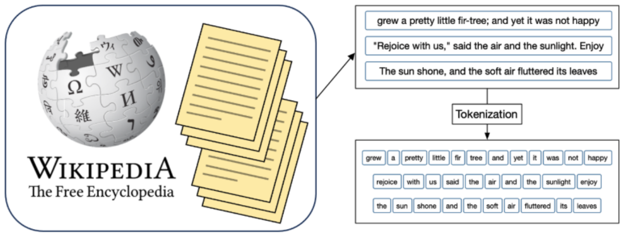

WIKIPEDIA
The Free Encyclopedia
grew a pretty little fir-tree; and yet it was not happy
"Rejoice with us," said the air and the sunlight. Enjoy
The sun shone, and the soft air fluttered its leaves
Tokenization
grew a pretty little fir tree and yet it was not happy
rejoice with us said the air and the sunlight enjoy
the sun shone and the soft air fluttered its leaves

Figure 1: The Process of Tokenization (Credit: Towards Data Science)

2.2 One way to tokenize: Byte Pair Encoding

So how do we create our subword vocabulary?

As mentioned, the key idea is that we want common chunks of characters to be their own elements. Those texts will then be the building blocks for less common strings. The **Byte-Pair Encoding Algorithm** is a method for defining V from a finite dataset. The method is simple:

1. Define a max size of $V =$ say $52k$.
2. While the size of $|v| < max$:
   - tokenize my document using current V
   - count most common adjacent tokens
   - concatenate the most common adjacent pair of elements into a string and add it to vocabulary
3. Repeat until $V == size$.

2.3 Problems in subword tokenization

However, subword tokenization still has some problems.

2.3.1 Language Imbalance

Since subword tokenization is a compression strategy, then it will do a great job at compressing the text it was trained on. The tokenizer is only as good as its inputs. Some issues could be:

- For a training corpus that is mostly English, the documents that are English will be split into a few large subword tokens (since we combine the most common ones). Other languages (such as Mandarin) will not be as common in the dataset and thus will be represented as many small subword tokens.
- Since the vocabulary is fixed, the languages with the most data will compete and get the most slots.

- While a longer common string is great, it is hard to maintain consistent context over a common long string. Ex: ”I went to the bank.”
- Long sequences have high compiational costs.
- Common words in the training dataset may not accurately resemble common words in spoken language or on the Internet. Ex: ”SolidGoldMagikarp” came up often in GPT-2. Why? That was the name of a really active Reddit user which was the main dataset it was trained on! [2]

2.3.2 Number Representations

Subword tokenization is not great for numbers. Consider the example:

```
"2005" + "6" = 2011
```

2005 is both a year and a quantity. How does the model know how to categorize it? Does the model represent a vocabulary element for 2005 (the number) and 2005 (the year)?

2.3.3 Unicode

A character is not exclusively one of the 26 English letters. Rather, a character is defined (in Python3) as all unicode strings. This creates a chicken and the egg problem:

- If we tokenize our entire vocabulary as all 150k unicode characters $\rightarrow$ we lose the inclusion of the common sets of strings we wanted to definitely have in our vocabulary.
- If we only tokenize the ”common” unicode characters $\rightarrow$ if the model sees a unicode character not in its vocabulary it will be unable to recognize it.

2.3.4 Size of Vocabulary

Of course, the larger the vocabulary the higher the coverage and the higher the cost in computing probabilities per token. A smaller vocabulary has longer sequences of tokens with more meaning, but bad scaling and potentially worse generation.

2.4 Summary

In order to be able to build representations of a large body of text, we need to create a vocabulary. We create a vocabulary with **tokenization**, a method of splitting apart small (common) chunks of text so when combined with other small chunks can form (larger) words with meaning. While there are many

[2]See: SolidGoldMagikarp (plus, prompt generation) on LessWrong

Model	**Provider**	**Vocab Size**	**Tokenization Method**
GPT-2 (2019)	OpenAI	50,257	Byte Pair Encoding (BPE)
GPT-3 (2020)	OpenAI	∼100,000	Modified BPE (100k_base)
GPT-3.5 Turbo	OpenAI	∼100,000	Modified BPE (100k_base)
LLaMA 3 (2024)	Meta	128,256	BPE-style subword tokenization
Mistral (e.g., 7B)	Mistral AI	∼32,000	Subword / (32k vocab + controls)
BERT (base)	Google	30,522	WordPiece tokenizer (subword)
Jurassic-1	AI21 Labs	250,000+	Subword (word-parts)
Gemini	Google	∼256,000	SentencePiece-style subwords

Table 1: Vocabulary sizes and tokenization methods for representative large language models.

methods of doing tokenization, one of the most common is Byte Pair Encoding. We do this on a huge corpus of data so our output can be the finite vocabulary that the model can use to generate probabilities on.

3 Neural Networks

So far, we've discussed learning from text and making vector representations of words. Now we discuss better building blocks for learning from text.

3.1 Sequence Representation

We model the next-token probability distribution $p(\cdot|x_{<i})$ by:

1. Computing a representation of the prefix $x_{<i}$, (Lecture 1)
2. Projecting that representation to the space of the vocabulary, (Lecture 2)
3. Normalizing to a probability distribution using the softmax function:

$$p(\cdot|x_{<i}) = \text{softmax}(Uh_{<I})$$

Where:

$$h_{<i} = f(x_1, \ldots, x_{i-1})$$

And U is the "unembedding matrix" that transforms the final high-dimensional hidden state back into token logit scores (probabilities) for the entire vocabulary, essentially performing the reverse of the input embedding process to predict the next word.

The remaining question is how to construct the prefix representation $h_{<i}$? A simple approach is to proceed as follows:

1. Average the embeddings
2. Add in positions of each word
3. Add a non-linearity to bind together variables
4. Add another non-linearity to bind together tuples of words

3.1.1 Vector Averages

We represent each token w_i by an embedding vector:

$$e_{w_i} \in \mathbb{R}^d$$

For example, if $d = 4$, then:

$$e_{\text{Columbia}} = [1.2, -1.03, 0.44, 2.17]$$

These numbers are learned parameters. During training, the model adjusts them so that tokens used in similar contexts end up with similar vectors. These embeddings are then stored in a matrix:

$$E \in R^{|V| \times d}$$

There are $|V|$ rows and d columns. Thus, the row $E[w]$ corresponds to the embedding of token $w \in V$.

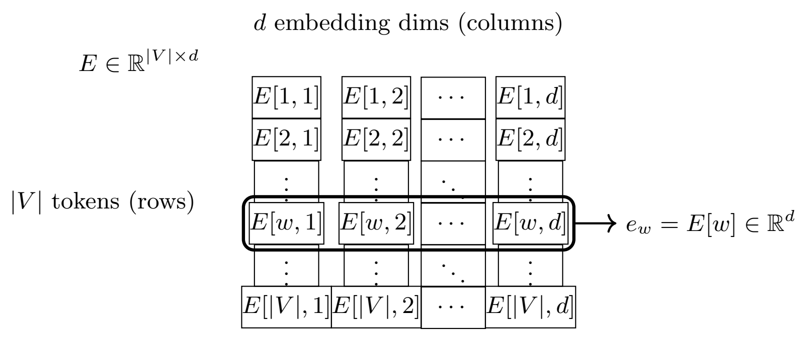


Figure 2: Embedding matrix E: each row stores the d-dimensional vector for a token; looking up token w returns row $E[w] = e_w$.

We can then average all word embeddings as:

$$h_{<i} = \frac{1}{i-1}\sum_{j=1}^{i-1} e_{x_j}$$

to obtain a single vector summarizing the preceding tokens. How does averaging work? Consider two sentences:

```
Benny ran to Abe.
Abe ran to Benny.
```

Both sentences contain the same set of tokens, so their averaged embeddings are identical. While this method is cheap and parallelizable, its permutation invariance prevents it from encoding word order, which is often essential for meaning. For this reason, we need to add positions.

3.1.2 Adding Positions

"You shall know a word by the company it keeps."

– J.R. Firth

Consider a sequence of words, $w_1, w_2, \ldots, w_{i-1}$. They can be ordered as simply as $1, 2, \ldots, i-1$. We define a maximum sequence length T, and learn a positional embedding matrix $P \in \mathbb{R}^{T\times d}$. for each embedded position. Then to represent a word w_i at position i:

$$E_{w_i} + P_i$$

Thus, in the two sentences above we have:

$$\text{Benny} \to P_1 + E_{\text{Benny}}$$

$$\text{Benny} \to P_5 + E_{\text{Benny}}$$

This way, the representation of "Benny" differs depending on its position. This can be generalized to all sentences where a position of a word and its surroundings imply its meaning. Our average embeddings end up being:

$$h_{<i} = \frac{1}{i-1} \sum_{j=1}^{i-1} \big(E[x_j] + P_j \big)$$

3.1.3 Non-Linearities

But won't the averages remain the same? Due to the law of additivity, there is nothing *tying* each word to each position. Let's see why:

$$\begin{aligned} h_{<i} &= \frac{1}{i-1} \sum_{j=1}^{i-1} (E[x_j] + P_j) && (1) \\ &= \frac{1}{i-1} \sum_{j=1}^{i-1} (E[x_j]) + \frac{1}{i-1} \sum_{j=1}^{i-1} (P_j) \qquad \text{(due to additivity)} && (2) \end{aligned}$$

How do we combine the information each word and position *before* we average them? We introduce **non-linearities**.

Non-linearities are "element-wise" functions applied independently to each dimension of a vector. For example:

- The ReLU function: $f(x) = max(x, 0)$
- Sigmoid function: $\sigma = \frac{e^x}{1+e^x}$

We use $\sigma(\cdot)$ to denote a generic element-wise non-linearity. We don't really care which non-linearity to use, so for generality we call it the function $\sigma(E_{w_i} + P_j)$. Applying a non-linearity constructs features that can no longer be decomposed into a sum of their parts. For example:

$$u = [1, -1] \quad \text{and} \quad v = [-3, 4]$$

$$[1, -1] + [-3, 4] \to u + v = [-2, 3]$$

$$ReLU(u + v) = ReLU([-2, 3]) \to [0, 3] \neq u + v$$

This shows that applying a non-linearity before averaging prevents the representation from being decomposed into separate word and position averages. The non-linearity σ prevents the word and position information from being separated after aggregation.

3.1.4 Non-linearities for token representations

But if we're just averaging, how do we express words that depend on other words in a sequence? For example:

```
I love movies, don't ...
I don't love movies,
```

How are we supposed to model dependencies such as negation? This motivates the need for representations that go beyond simple averaging. We want the words to interact "non-linearly" with eachother, so we use the same trick as how we combined the word embeddings with positions:

$$h_{<i} = \sigma\left(\frac{1}{i-1}\sum_{j=1}^{i-1}\sigma(E[x_j] + P_j)\right)$$

Which becomes our "deep network". Our non-linearity #1 combines each word with position, and non-linearity #2 increases the expressive capacity of the aggregated representation.

3.1.5 Summary

To summarize, our model:

1. Embeds each token with both semantic and positional information.
2. All words are averaged
3. A simple non-linear function (ie. sigmoid) non-linearly combines those components.
4. We form a distribution over the vocabulary.

While this increases expressive power beyond linear averaging, it still cannot model explicit token–token dependencies. To capture such interactions, we will introduce attention in Lecture 7.

3.2 Optimization

Now that we've learned how non-linearities allow for the construction of representations of sequences, we want to focus on how the neural network actually learns from them. Optimization is the algorithms that choose the weights of a neural network in order to perform well on a given objective.

The key goal of optimization is to search over the space of all possible values of the parameters of our network to find the values that minimize the loss (which means the network is performing well). Recall that stochastic gradient descent updates parameters of a network as:

$$\theta_t = \theta_{t-1} - \alpha\nabla_\theta\ \mathcal{L}$$

Where $\mathcal{L}$ is the loss function (direction) and α is the learning rate. Think of $\mathcal{L}$ as "locally, in what direction do we want to go in" and α as the step size. In a simple quadratic, the $\mathcal{L}$ can be as simple as its first derivative, whereas in a function with several variables and dimensions things get much more complicated very quickly. This matters because we want to train very deep networks to create parameters that we can train on.

How do we best optimize this? We introduce the **Adam-variant of gradient descent**. The key idea is that we combine the benefits of momentum and adaptive learning rates to efficiently train the model.

$$m_t = \beta m_{t-1} + (1-\beta)\nabla_\theta \mathcal{L}$$

$$v_t = \beta v_{t-1} + (1-\beta)(\nabla_\theta \mathcal{L})^2 \quad \text{(element-wise)}$$

So then:

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{v_t}+\epsilon} m_t \qquad \text{(simplified Adam update)}$$

Whats the idea here?

- m_t is the exponentially decaying average of past gradients (momentum).
- v_t is the exponentially decaying average of past squared gradients.
- β controls how much past information is retained.
- ϵ is added for numerical stability.

The key takeaway is that optimization choices strongly affect whether training is stable, efficient, and scalable.

3.3 Neural Network Architecture

How do models actually learn at each step? At a high level, a neural network is a parameterized function that maps an input representation to an output by composing simple operations. Concretely, these operations consist of linear transformations followed by non-linear functions. Given an input vector $x \in \mathbb{R}^d$, a basic neural network layer takes the form

$$f(x) = \phi(Wx + b),$$

where W and b are learned parameters and ϕ is a non-linear activation function (we'll be using a similar function).

While a single such transformation is limited in what it can represent, composing many of them allows the network to model increasingly complex functions. The specific way these transformations are composed—how layers are arranged, how information flows between them, and how intermediate representations are combined—is referred to as the network architecture. These architectural choices determine both the expressive power of the model and how easily it can be optimized.

In this section, we introduce the architectural building blocks that enable neural networks to form rich, trainable representations, setting the stage for modern sequence models.

3.3.1 Feed Forward Layers

The problem:
Averaging and linear maps are too weak to capture real language structure.

The solution:
Feed forward layers are a function of the form:

$$f(h) = A\sigma(Bh)$$

Where:

- $h \in \mathbb{R}^d$ denotes a learned representation of the input sequence.
- $B \in \mathbb{R}^{p \times d}$ is the "first weight matrix", where input $d \to$ hidden p.
- σ is the non-linearity for expressive power.
- $A \in \mathbb{R}^{q \times p}$ is the "second weight matrix", where hidden $p \to$ output q.

We apply multiple identically sized feed forward layers in sequence to add learning capacity to a network. How does this work?

1. We have the prefix h and we apply a linear transformation on it B to receive a new transformed matrix.
2. Bh maps h into our new space p.
3. We apply the linearity $\sigma(Bh)$ for expressivity
4. After another linear transformation $A\sigma(Bh)$, we get our output q

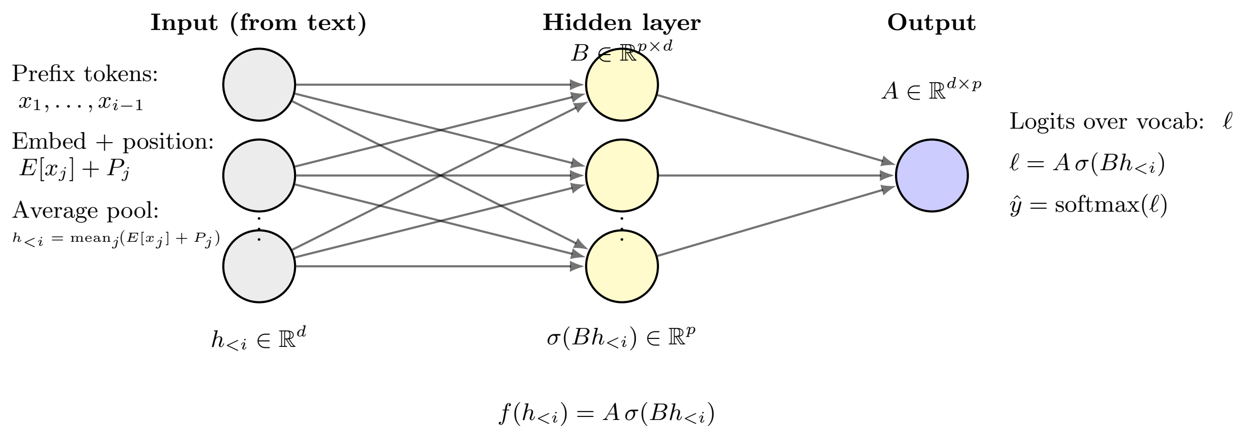
We can visualize this as:
Since this is the way our network learns, our prefix function can be redefined from:

$$h_{<i} = \sigma\left(\frac{1}{i-1}\sum_{j=1}^{i-1}\sigma(Ex_j + p_j)\right)$$

To:

$$h_{<i} = FF\left(\frac{1}{i-1}\sum_{j=1}^{i-1}FF(Ex_j + p_j)\right)$$

As σ is a placeholder for any complex non-linear function.



Input (from text)
Hidden layer
Output
$B \in \mathbb{R}^{p\times d}$
Prefix tokens: $x_1, \dots, x_{i-1}$
Embed + position: $E[x_j] + P_j$
Average pool: $h_{<i} = \text{mean}_j(E[x_j] + P_j)$
$A \in \mathbb{R}^{d\times p}$
Logits over vocab: ℓ
$\ell = A\,\sigma(Bh_{<i})$
$\hat{y} = \text{softmax}(\ell)$
$h_{<i} \in \mathbb{R}^d$
$\sigma(Bh_{<i}) \in \mathbb{R}^p$
$f(h_{<i}) = A\,\sigma(Bh_{<i})$

Figure 3: NLP feed-forward network: tokens are embedded and position-encoded, pooled into $h_{<i}$, then transformed by a two-layer feed-forward network $f(h_{<i}) = A\sigma(Bh_{<i})$ to produce logits ℓ over the vocabulary.

3.3.2 Residual Connections

The Problem:
When stacking non-linearities in a large feature forward (FF) model, the gradients can go to 0 or ∞.

The solution:
Residual connections adds the input to the function every time it makes a learning go around.

$$\tilde{f}(h) = h + f(h)$$

This ensures that the network can always behave like the input identity if needed.

3.3.3 Gating

The Problem:
It is not clear if we should be updating the value of every vector in every learning phase.

The Solution:
Gating helps control the flow of effects through a network. If there is some function v that modifies our prefix h:

$$h \leftarrow h + v$$

v is always included! How do we make sure we *want* to add it? What if we only add v depending on some properties of h? This is what we refer to as gating. There are a few different kinds of gates:

- Scalar gate: $h \leftarrow h + \sigma(h^T v)v$
  We only add v to the extent its similar to h.

- Element-wise gate (per feature): $h \leftarrow h + \sigma(h) \odot v$
  We add v to h more if its very negative and less if its very positive.

- Forget / read gate: $h \leftarrow \sigma(-h) \odot h + \sigma(v) + v$
  Here we see how much the network should delete ("forget") in every pass, and how much it should add ("read") in every pass.

3.4 Summary

In this section, we built up the core machinery needed to represent and learn from sequences with neural networks. We started by embedding tokens as vectors and showed how simple aggregation methods like averaging produce fixed-dimensional representations, while also making clear what information they discard. By adding positional information and non-linear transformations, we saw how models can encode order and avoid collapsing representations into purely linear summaries. We then introduced the architectural components that make these representations useful in practice—feed-forward layers to increase expressiveness, residual connections to stabilize optimization, and gating mechanisms to control how information is updated. Alongside this, we discussed optimization procedures that allow these models to be trained efficiently in high-dimensional, non-convex settings.

At the same time, this section highlighted an important limitation: building expressive representations is not the same as solving language tasks. The models described here transform and aggregate information, but they do not yet specify:

1. what should be predicted,
2. how predictions are evaluated,
3. and which parts of a sequence should influence one another for a given decision.

In the next lectures, we move from model construction to application, introducing concrete tasks, evaluation frameworks, and machine translation as a central example. This shift will make clear why certain architectural choices succeed or fail in practice, and why effective sequence modeling ultimately requires mechanisms that can selectively focus on and relate different parts of an input when producing outputs.

4 Tasks and Evaluations

Now that we've defined what we want our system to do, namely:

1. Learn language
2. Understand language
3. Generate language

how do we test and know how well our systems work? There are a few options, in order of difficulty:

1. Try it - testing purely on vibes and intuition.
2. Try it - test on a verifiable output.
3. Create tasks - create a set of problems you want to solve.

This is really important for answering the questions a company wants to answer:

1. Should I release my model?
2. I have developed models A and B. Is $A > B$?

Tasks will be the bulk of our evaluations. For example, consider machine translation from English to Portuguese. Let V_{en} and V_{pt} be the English and Portuguese vocabularies. An input sentence is a sequence

$$x_{1:T} \in V_{\text{en}}^T,$$

and the model outputs a Portuguese sequence

$$\hat{y}_{1:m} = f(x_{1:T}) \in V_{\text{pt}}^m.$$

In principle, V_{en}^* contains arbitrary strings (including rare symbols), but in practice we only evaluate translation on a realistic input distribution. Let

$$x_{1:T} \sim D$$

where D is, for instance, the distribution of well-formed English sentences (e.g., news text). For each input $x_{1:T}$, we assume a reference translation $y_{1:m}$. Thus a translation task instance is:

$$x_{1:T} \sim D \tag{3}$$
$$y_{1:m} \sim p(\cdot \mid x_{1:T}) \approx \text{(reference)} \tag{4}$$

For example:

```
x:  "I love to play soccer"
y:  "eu amo jogar futebol"
```

For other tasks, the same template applies: sample an input from D, and compare the model's output $\hat{y}$ to an expected output y. For instance, in automated theorem proving:

```
x:  mathematical conjecture
y:  a proof of x (e.g., in Lean)
```

But what if no one proved that mathematical conjecture before? What if no one speaks Portuguese? For tasks like arithmetic or symbolic reasoning, evaluation is straightforward—outputs can be checked exactly. Language, by contrast, admits many valid answers. A translation can be wrong, awkward, or perfect, with no sharp boundary between them. This ambiguity forces us to operationalize "correctness." A common abstraction is to view a task as a mapping:

$$x \mapsto \bar{y}$$

where x is the input and $\bar{y}$ is the models output. We then compare $\bar{y}$ to a reference answer y, reducing to a scoring function:

$$g(x, \bar{y}, y)$$

that measures how close the models output is to the correct answer.

Why does it make sense to check if its an exact match? Entropy! One way to mitigate entropy is with multiple choice. Consider that any word can have multiple synonyms in the translated language, so predictions can be "matched". For multiple choice, there is 0 entropy over the possibility of correct answers.

4.1 Human Evaluation

So far we've been operating on the model of just general evaluation. However, to narrow down our analysis, one of the main ways to evaluate is with human evaluators. We could simply ask ourselves if a response is "good." For example:

$$x : \quad g(x, \bar{y}) \mapsto \{1, 2, 3, 4, 5\} \tag{5}$$

$$y : \quad \text{"How good is this translation on a Likert scale?"} \tag{6}$$

There are obvious problems:

1. Bias (dialects, human variation, etc.)
2. Lack of context
3. Lack of knowledge of Likert scale
4. Its often easier for people to *compare* outputs than to score absolutely. A potential solution for this is to generate twice from the model: $g(x, \bar{y_1}, \bar{y_2}, y)$ thus allowing for comparison.

How do we make human evaluations better?

- Count major errors and minor errors
- Create a formula like 5 major errors $*$ 1 minor error

However still a slow process and hard to replicate.

4.2 BLEU Scores (2002)

BLEU (Bilingual Evaluation Understudy) is one of the earliest and most influential automatic metrics for evaluating machine translation. The core idea is simple: the quality of a translation is measured by how closely it matches a human reference translation. Informally, the closer a model's output is to what a human would write, the better the translation is assumed to be.

Formally, let $\hat{y} = f(x)$ be the model's predicted translation of an input sentence x, and let y be a reference (human) translation. A natural first question is:

> What fraction of the words in $\hat{y}$ also appear in y?

However, this naive approach immediately runs into several problems:

1. It is unclear what counts as a "word": dictionary words, tokens, or characters.
2. It ignores word order entirely.
3. It does not account for synonyms.
4. If $\hat{y}$ is a permutation of y, it would be scored as perfectly correct.

BLEU addresses these issues by rewarding overlap not just at the word level, but at the level of *contiguous word sequences*, or *n-grams*. Intuitively, matching longer phrases is stronger evidence of a good translation than matching isolated words. Let the reference translation be

$$y = (y_1, y_2, y_3, y_4).$$

We can extract:

$$\begin{aligned} \{\bar{y}_1, \bar{y}_2, \bar{y}_3, \bar{y}_4\} &\quad \text{(unigrams)} \\ \{(\bar{y}_1, \bar{y}_2), (\bar{y}_2, \bar{y}_3), (\bar{y}_3, \bar{y}_4)\} &\quad \text{(bigrams)} \\ \{(\bar{y}_1, \bar{y}_2, \bar{y}_3), (\bar{y}_2, \bar{y}_3, \bar{y}_4)\} &\quad \text{(3-grams)} \end{aligned}$$

BLEU typically considers n-grams up to $n = 4$. As a concrete example, consider:

```
y:   Eu amo a pizza
ŷ:   Eu gosto da pizza
```

The resulting n-gram overlaps are:

n-gram	overlap
1	$\frac{1}{2}$
2	$\frac{1}{3}$
3	0
4	0

Table 2: n-gram overlap for the example translation

Although this translation does a poor job of capturing the exact meaning, it is still easily understandable to a human speaker. BLEU captures this intuition by assigning partial credit for preserving local structure, even when higher-order structure is missing.

For nearly two decades, BLEU was the dominant evaluation metric in machine translation and broader NLP research. Influential models such as *Attention Is All You Need* relied on BLEU scores to demonstrate progress and to distinguish weak models from strong ones. Despite its limitations, BLEU played a central role in making large-scale evaluation of language generation practical.

4.3 Summary

At a high level, evaluating language models reduces to evaluating the tasks we ask them to perform. Framing a task as a mapping from an input distribution to an expected output gives us a clean abstraction, but it immediately exposes a deeper difficulty: for most language tasks, there is no single "correct" answer. Unlike arithmetic or symbolic reasoning, language admits ambiguity, paraphrase, and multiple valid outputs, forcing evaluation to rely on proxies rather than exact correctness.

5 Applying the Concepts: Translation

We now apply the ideas from the previous sections to a concrete task: machine translation. Our goal is to translate a sentence from a source language into a target language, using text-only inputs and outputs. We assume access to a dataset

$$D = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m},$$

where each $x^{(i)}$ is a sentence in the source language and each $y^{(i)}$ is its corresponding translation in the target language. In what follows, we assume m is large (e.g., $m \approx 100,000$).

For tokenization, To operate on text, we first convert sentences into sequences of tokens. We use Byte Pair Encoding (BPE), which balances vocabulary size with the ability to represent rare words.

BPE is learned jointly over source and target languages, which:

- creates a shared vocabulary,
- enables partial lexical overlap across languages,
- reduces the impact of rare or unseen words.

After tokenization, a source sentence becomes:

$$x = (x_1, \dots, x_T), \quad x_t \in V,$$

and a target sentence becomes:

$$y = (y_1, \dots, y_m), \quad y_j \in V,$$

where V is the shared vocabulary. Our objective is to model the conditional distribution:

$$p(y \mid x),$$

the probability of a target sentence given a source sentence. Using the chain rule, we factor this distribution autoregressively:

$$p(y \mid x) = \prod_{j=1}^{m} p(y_j \mid y_{<j}, x).$$

We prepend a special `[START]` token to the target sequence and terminate generation with an `[END]` token. We then specify how the conditional distribution $p(y \mid x)$ is modeled. As in language modeling, we generate the target sentence one token at a time, conditioning on both the source sentence and the previously generated tokens. The probability of the next token is given by:

$$p(y_j \mid y_{<j}, x) = \text{softmax}(W h_j + b),$$

where W and b are learned parameters. Training proceeds by maximizing the likelihood of the reference translations in the dataset. Equivalently, we minimize the negative log-likelihood:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{m} \sum_{j=1}^{|y^{(i)}|} \log p_\theta\left(y_j^{(i)} \mid y_{<j}^{(i)}, x^{(i)}\right).$$

This objective is optimized using gradient descent. Aside from conditioning on the source sentence x, this training objective is identical to that of a standard language model.

5.1 Generation

Now that we have some $p(y|x)$, how do I make my function that actually translates? We introduce **greedy decoding**. Simply, iteratively, pick the most likely next word. We begin by predicting the first target token, conditioning only on the source sentence:

$$p(y_1 \mid x_{1:T}).$$

The model produces a distribution over the vocabulary via a softmax:

$$\text{softmax}(Eh_{x_{1:T}}) = [\text{I} = 0.7,\ \text{we} = 0.2,\ \ldots,\ \text{movies} = 0.01].$$

We then select the most likely token:

$$\hat{y}_1 = \arg\max_{w \in V} p(w \mid x_{1:T}).$$

Effectively asking, which $w \in V$ has the highest probability? This process is repeated autoregressively. At step j, we condition on the source sentence and all previously generated tokens:

$$\hat{y}_j = \arg\max_{w \in V} p(w \mid x_{1:T}, \hat{y}_{<j}).$$

Decoding continues until a termination condition is met. Once a token is selected, it is appended to the output sequence and treated as fixed. The next prediction depends entirely on the previous choices, and at each step we again select the most likely continuation.

After decoding, the sequence of predicted tokens

$$\hat{y}_{1:m}$$

is converted back into text through a **detokenization** process.

Q: Instead of estimating next word probabilities, why don't we estimate the next *sequence* probability?
A: Then you need to guess all the previous words! Too much guessing. Autoregressive decoding avoids this by committing to one token at a time.

How does the model end the translation?

J'aime les filmes **[START]** $\rightarrow$ I like movies **[END]**

[END] is calculated in the probabilities for every potential next word!

We then evaluate the model as suggested in the previous lecture, iterating on the model and changing $hw_{<t}$ to better neural networks, then checking to see if the BLEU score went up.

5.2 Summary

- A translation task is framed as learning a conditional distribution $p(y \mid x)$, where a source sentence x is mapped to a target sentence y.
- Training data consists of parallel sentence pairs $(x^{(i)}, y^{(i)})$, drawn from a realistic input distribution rather than all possible strings.
- Text is tokenized (e.g., using BPE), producing sequences of discrete tokens from a shared vocabulary across source and target languages.
- The model factorizes the conditional distribution autoregressively:

  $$p(y \mid x) = \prod_j p(y_j \mid y_{<j}, x),$$

  allowing translation to be treated as conditional language modeling.
- Model parameters are learned by maximizing the log-likelihood of reference translations using cross-entropy loss and gradient descent.
- At inference time, the model must generate a translation without access to the reference output.
- Generation begins with a special `[START]` token and proceeds token by token using greedy decoding, selecting the most likely next word at each step.
- Generation terminates when the model emits the `[END]` token, which is included in the vocabulary and scored alongside all other tokens.
- The resulting token sequence is detokenized to produce readable text.
- Model quality is evaluated by comparing generated translations to human references using automatic metrics such as BLEU.
- Improvements are driven by iterating on model architecture, training, and decoding strategy, and tracking whether evaluation metrics improve.

This puts us in good shape to move on to the next lecture, how to *scale* these systems.

6 GPUs and Parallelizable Architecture

Deep networks are trained on hundreds of billions of operations per second. But the goal isn't *more math*, rather its *how fast we can do that math.* We've learned that a good model architecture must have:

- **Expressivity** - how it can represent complex functions.
- **Efficiency** - how it can learn quickly in terms of data and time.

In this lecture, our focus is efficiency. We can divide efficiency into two parts:

1. **Data efficiency** - how well does the model learn from limited samples?
2. **Time efficiency** - how quickly can the model reach good performance on a limited hardware budget?

At a high level, efficiency in modern language models is achieved by structuring computation so that large parts of it can be executed in parallel. GPUs excel at applying the same operations to many vectors at once, but this requires the model architecture to expose parallel structure rather than long chains of sequential computation. To answer these questions, we introduce stacking components.

6.1 Stacking Components

The core challenge is to represent everything the model has seen so far using a fixed-size vector that can be updated efficiently. Rather than repeatedly reprocessing the entire prefix, the model maintains a running representation of context that summarizes the relevant information from previous tokens. Recall the prefix function and probability setup:

$$p(\cdot|x_{<i}) = \text{softmax}(Uh_{<i})$$

where $U \in \mathbb{R}^{|V|\times d}$ is the unembedding matrix and:

$$h_{<i} = f(x_1, \ldots, x_{i-1})$$

is the prefix representation - a vector summarizing the sequence so far. The goal of the network is to learn f, i.e. how to map a prefix of text to a good contextual representation $h_{<i}$.

A single transformation is generally insufficient to capture the hierarchical structure of language. Words combine into phrases, phrases into clauses, and clauses into meaning. To capture these increasingly abstract patterns, models apply the same basic transformation repeatedly, refining the representation at each step.

How does the model learn f? It follows a **stack architecture** - multiple stacks (or ”layers”) built on top of one another. Each layer (ℓ) takes the previous layer’s ($\ell - 1$) output, plus some contextual information from the previous time step, then runs that through a feed-forward network (FF). Here, ℓ indexes depth (layers), while t indexes position in the sequence. Stacking operates across layers, while context flows across time. We express this as:

$$x_{t-1}^{(\ell)} = FF(x^{\ell-1} + \mu_{t-1}^{\ell-1})$$

where:

- $x_{t-1}^{(\ell)}$ is the vector representing the prefix up to word t in the previous layer.
- $\mu_{t-1}^{(\ell-1)}$ is information coming from the previous token’s representation (so the model remembers sequence structure).
- The sum $x^{\ell-1} + \mu_{t-1}^{\ell-1}$ merges current and contextual info a residual connection.
- $FF(\cdot)$ non-linearly transforms this combined input into a new space.

And we can visualize it as such:

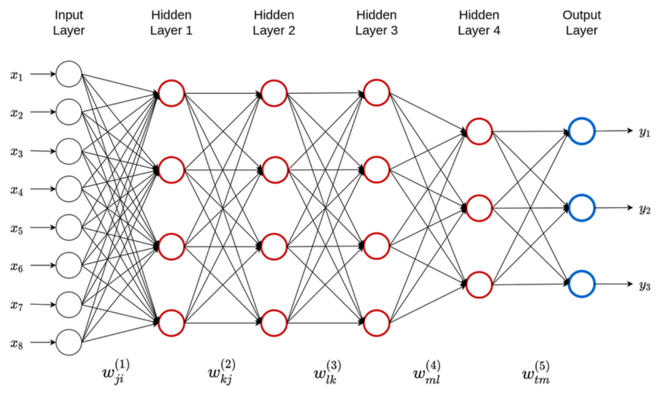


Figure 4: Example of stacking components

6.2 Matrix Multiplication

What goes on in each of these blocks? Each layer learns through repeated matrix multiplications and non-linearities that build more expressive representations on each stack.

Each matrix multiplication teaches new properties by changing the geometry of the space the model lives in. It does that by:

- Rotation $\rightarrow$ mixes dimensions in a new way.

- Scaling and stretching $\rightarrow$ amplifies some directions, downplays others.
- Projection $\rightarrow$ compresses irrelevant dimensions.

Recall that our feed-forward network is literally a series of matrix multiplications. Previosly we defined it as:

$$f(h) = A\sigma(Bh)$$

Where:

- $h \in \mathbb{R}^d$ is a representation of the prefix.
- $B \in \mathbb{R}^{p \times d}$ is the "first weight matrix", where input $d \rightarrow$ hidden p.
- σ is the non-linearity for expressive power.
- $A \in \mathbb{R}^{q \times p}$ is the "second weight matrix", where hidden $p \rightarrow$ output q.

And we can rewrite it as:

$$FF(x) = W_2\sigma(W_1x + b1) + b2$$

where:

- $W_1 \rightarrow$ rotates, stretches, projects to hidden space.
- $\sigma \rightarrow$ bends the space (non-linearly).
- $W_2 \rightarrow$ re-projects back.

Each layer then reuses and refines these transformations to build better understanding.

6.3 GPUs and TPUs

While the ideas of deep learning have been around for decades, they became practical when GPUs (Graphics Processing Unit) became prevalent because of the video game industry!

To simplify, tasks in a CPUs (Central Processing Unit) are computed serially, thus making them great for a few complex sequential tasks (like running an operating system). On the other hand, GPUs are specialized processors designed to perform many calculations simultaneously. Tasks that involve large-scale, repetitive calculations, such as gaming, video editing, 3D rendering, and machine learning are all optimized as they can be broken down into many parallel computations.

If you could represent functions as big matrix multiplications, then the GEMM can be computed very quickly because each dot product is independent of the computation.

For example:

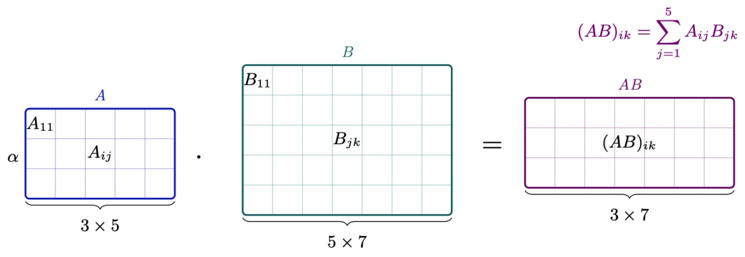


Figure 5: Matrix Multiplication - Credit to John Hewitt

While a CPU would calculate each dot product within each "square" sequentially, a GPU would be able to calculate them all in parallel!

6.3.1 Utilization: memory bandwidth and arithmetic intensity

There are two things we are always doing when computing operation:

1. The time we move bytes around and accessing memory, defined as T_{mem}
2. The time we actually do math, defined as T_{math}

Thus, our time for our program is:

$$T = \max(T_{\text{mem}}, T_{\text{math}})$$

where our program is **memory bound** if:

$$T_{\text{mem}} > T_{\text{math}}$$

and our program is **math bound**.

$$T_{\text{mem}} < T_{\text{math}}$$

To approximate T_{mem} and T_{math}, we use the following rough calculations:

$$\textbf{arithmetic intensity} = \frac{C_{math}}{C_{bytes}}$$

where C_{bytes} is the number of bytes we access for our computation and C_{math} be the number of floating point operations. We also create the:

$$\textbf{ops-to-byte ratio} \; = \frac{B_{math}}{B_{bytes}}$$

where B_{math} is the math bandwidth (ie. the speed at which we compute mathematical operations) and B_{bytes} our memory bandwidth (ie. the speed at which we access and move memory). We then get:

$$T_{\text{math}} = \frac{C_{math}}{B_{math}}$$

$$T_{\text{mem}} = \frac{C_{bytes}}{B_{bytes}}$$

We want operations that have a lot of math relative to bytes so we optimize computation (as models with more math have better predictions).

6.4 Summary

- Modern NLP models are computationally intensive due to large matrix multiplications arising from embeddings, attention, and feed-forward layers.
- GPUs are specialized for **massively parallel numerical computation**, making them well-suited for linear algebra operations such as matrix–matrix and matrix–vector products.
- Compared to CPUs, GPUs trade general-purpose control flow for **high throughput**, enabling thousands of lightweight threads to execute the same operation simultaneously.
- Training large language models is dominated by:
  - forward passes (computing activations),
  - backward passes (computing gradients via backpropagation),
  - parameter updates using gradient-based optimization.

7 Attention and Transformers

Previously we learned that we make the model more expressive, ie. more capable of understanding subtle patterns in language, by adding stacked, nonlinear transformations (matrix multiplications + activations + residuals). However there are still several issues with these recurrent neural networks.

7.1 Problems With Our Current Model

7.1.1 Linear Interaction Distance

At a high level, the problem is that information in an RNN can only travel locally, from one word to the next. If two words are far apart in the sentence, the model has no direct way to compare them—it must pass information step-by-step through every intermediate token. Take the sentence:

```
The weather in New York City in the fall is chilly although beautiful.
```

To decide what "beautiful" refers to, the model has to remember "weather." However, "weather" is many words away. So in an RNN:

1. The hidden state h_1 (for "the weather") updates to h_2, h_3 etc.
2. Each new h_t depends on the previous one:

$$h_t = f(h_{t-1}, x_t)$$

3. By the time we reach "beautiful" the information about "the weather" has passed through every intermediate hidden state. Think of broken telephone, whispering this to 10 people in line. So too, it will take $O(\text{sequence length})$ steps for distant word pairs to interact.

This is particularly a problem because long-distance dependencies fade. Each step slightly distorts or forgets earlier info, gradients become tiny (see the vanishing gradient problem), and the training can't preserve relationship between far-apart words. This model takes the assumption that language is strictly left-to-right, even though meaning doesn't always follow word order. More generally, RNNs impose a strong inductive bias: meaning is assumed to flow primarily from nearby words and in strict left-to-right order. Consider the following sentence:

```
There's really nothing like it.
We had a great time, ate good food, and enjoyed the weather.
New York City in the fall is the best.
```

Our existing model takes a rigid sequence bias and would not know that "nothing like it" refers to "New York City."

7.1.2 Lack of Parallelizability

This sequential dependency is not just a modeling limitation—it is a fundamental computational bottleneck. As h_t depends on h_{t-1}, you can't compute all time steps at once. You must wait for each word and step to finish before moving on to the next one, thus making it inefficient for GPUs which optimize best with parallelization. Future RNN hidden states can't be computed in full before past RNN hidden states have been computed. This inhibits training on very large datasets.

This is especially a prevalent as GPUs and TPUs have become more powerful over time, thus not justifying having a bottleneck in time cost.

7.2 Attention

The goal of attention is to let each word gather the information it needs from the rest of the sequence, producing contextualized vectors that feed into deeper layers and ultimately into the model's predictions. Concretely, attention removes the requirement that information flows through intermediate words. Each token is allowed to directly inspect every other token in the sentence and decide, for itself, which ones are relevant. This turns a long chain of dependencies into a single, global interaction step.

A good mental model for attention is that it performs "fuzzy lookup" in a key-value store. Instead of choosing a single value, attention takes a weighted average by "softly attending" to several keys depending on similarity.

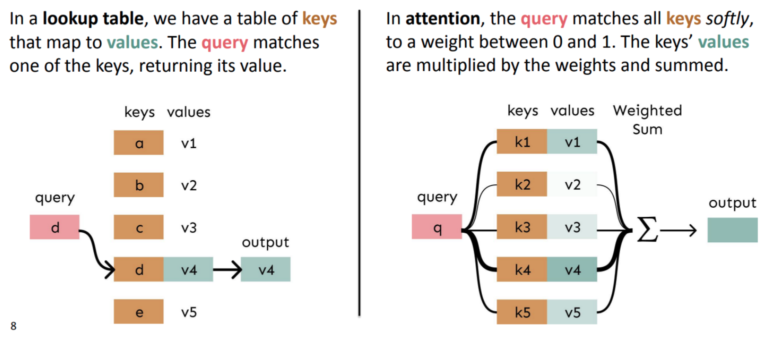

In a **lookup table**, we have a table of keys that map to values. The query matches one of the keys, returning its value.
keys
values
a
v1
b
v2
query
c
v3
d
d
v4
output
v4
e
v5
8
In **attention**, the query matches all keys *softly*, to a weight between 0 and 1. The keys' values are multiplied by the weights and summed.
keys
values
Weighted Sum
k1
v1
k2
v2
query
q
k3
v3
k4
v4
k5
v5
Σ
output

Figure 6: Attention Mental Model (Credit: John Hewitt)

To implement this idea, we give each word three different roles: one for asking what it needs, one for advertising what it contains, and one for contributing information. Formally, let $w_{1:n}$ be a sequence of words in vocabulary Z. For each w_i, let $x_i = Ew_i$, where $E \in \mathbb{R}^{d\times|V|}$ is an embedding matrix. We create three versions of each word by multiplying with learned matrices:

$$q_i = Qx_i \quad \text{(Query)}$$

$$k_i = Kx_i \quad \text{(Key)}$$

$$v_i = Vx_i \quad \text{(Value)}$$

Where $Q, K, V \in \mathbb{R}^{d \times d}$. For each token i we compute pairwise similarities between its key and all queries:

$$e_{ij} = q_i^T k_j$$

Recall that the dot product measures how well the query and key are aligned. We then normalize with softmax (so the weights sum to 1):

$$a_{ij} = \frac{\exp(e_{ij})}{\sum_j \exp(e_{ij})}$$

Then we compute output for each word as a weighted sum of values:

$$o_i = \sum_j a_{ij} v_i$$

The output of each o_i is a new vector for word i that depends on what it found important in the rest of the sentence. The new embedding for each word encodes not just the word itself, but its relationship to everything else. Intuitively, every word looks at all other words, decides which one matters most, and forms a new vector that's a weighted average of the others.

By doing so, the maximum interaction distance between two words is $O(1)$! Because there is no sequential dependency, ie. every token computes its attention scores to all other tokens in one operation, there is no need to walk through intermediate tokens! However, its important to note that $O(1)$ does not mean that computation is time constant (we still need to do a bunch of matrix multiplies). Rather, the *graph distance* does not grow with sequence length. Every token can "see" every other token in a single forward pass. Unlike RNNs, where information must traverse every intermediate hidden state, attention allows any two tokens to interact in a single layer.

7.2.1 Barriers and Solutions for Self Attention

A key consequence of this formulation is that self-attention is permutation-invariant: it has no built-in notion of word order. We've learned that attention connects every word to every other word using dot products of embeddings:

$$e_{ij} = q_i^T k_j$$

However, q_i and k_j come only from the word embeddings $x_i = Ew_i$. If we shuffle the words, we're getting the same sort of vectors but in a different order. There is no **position representation**! Mathematically, if we permute the sequence $x_1, x_2, \dots, x_n$, the numerator and denominator in our softmax function reorder in exactly the same way:

$$a_{ij} = \frac{\exp(e_{ij})}{\sum_j \exp(e_{ij})}$$

The solution to this is to inject positionional information. We define a position embedding $P_i \in \mathbb{R}^d$ and simply add a representation of the position of a word to its word embedding:

$$\tilde{x} = x_i + P_i$$

Now each token's vector carries both its identity (the word embedding) and its location (the position embedding). From this point on, queries and keys are computed from $\tilde{x}_{ij}$.

Even with positional information, stacking attention layers alone does not increase expressivity: attention is still a linear operation over values. The next problem is that if you stack multiple attention layers after another without any non-linearity you don't actually make the model more expressive. As of now, its just weighted averages! The solution for this is to apply a feed-forward network to post-process each output vector. The intuition is that the feed-forward network processes the result of attention:

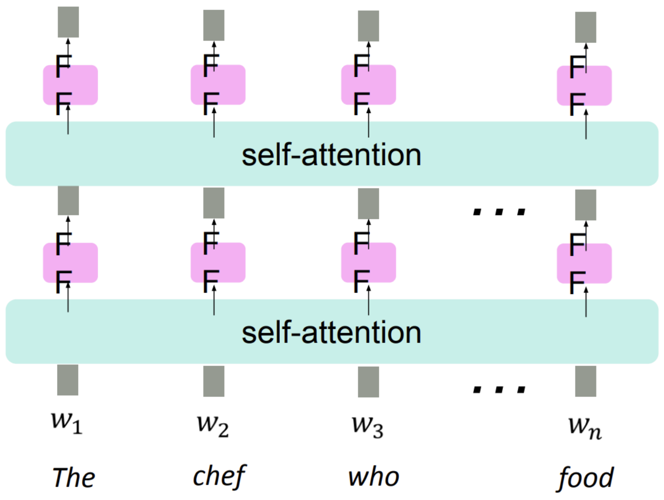


Figure 7: Feed-Forward Layer after each Attention Block (Credit: John Hewitt)

Mathematically, this is simply:

$$h_{FF} = W_2 * \text{ReLU}(W_1 h_{attention} + b_1) + b_2$$

Where:

- After each attention block, you have one vector per token $h_{attention} \in \mathbb{R}^d$
- $W_1 \in \mathbb{R}^{5d \times d}$ expands the dimension by 5.
- b_1 is a bias term.
- $\text{ReLU}(x)$ is the elementwise non-linearity.
- $W_2 \in \mathbb{R}^{d \times 5d}$ projects back down to the original dimension.
- b_2 is the final bias.

By doing so, the model **adds expressivity through the classic non-linearity** deep learning magic.

In language modeling, the task is inherently causal: when predicting the next word, the model should not have access to future words. However, self-attention is symmetric by default—every token can attend to every other token. We need to ensure we don't "look at the future" when predicting a sequence. How can this happen? In language modeling, we want to predict the next word given all previous words:

$$p(w_t | w_1, w_2, \ldots, w_{t-1})$$

so the model should only use words up to $t-1$ to predict the next word. However, attention allows every word to see every other word (by design!). This opens up the model to see future words when training current words. For example:

```
Zuko made his uncle tea
```

At each word, the model should be trained to predict the next word given only the previous words before the current one. During training, when the model tries to predict the word "tea" it uses the entire previous sequence. This means that the model isn't really *predicting* "tea", its *copying* it. So during inference time, the model won't work because it won't have the future words it was trained on!

The way to mitigate this is by "masking" the future words during training by setting attention scores to $-\infty$:

$$e_{ij} = \begin{cases} q_i^\top k_j, & \text{if } j \leq i \\ -\infty, & \text{otherwise} \end{cases}$$

This ensures that token i can only attend to itself and earlier tokens.After applying the softmax, any entry whose score was set to $-\infty$ receives probability zero. In practice, we approximate $-\infty$ with a large negative constant (e.g. $-10,000$. Visually we can depict this as:

$$
\begin{array}{c|ccccc}
 & \text{Zuko} & \text{made} & \text{his} & \text{uncle} & \text{tea} \\
\hline
\text{Zuko} & & -\infty & -\infty & -\infty & -\infty \\
\text{made} & & & -\infty & -\infty & -\infty \\
\text{his} & & & & -\infty & -\infty \\
\text{uncle} & & & & & -\infty \\
\text{tea} & & & & &
\end{array}
$$

Figure 8: Diagram of Future Masking (Credit: John Hewitt)

So words in each row can only "look" at non-zeroed out words!

To summarize what we have so far, we have the following steps:

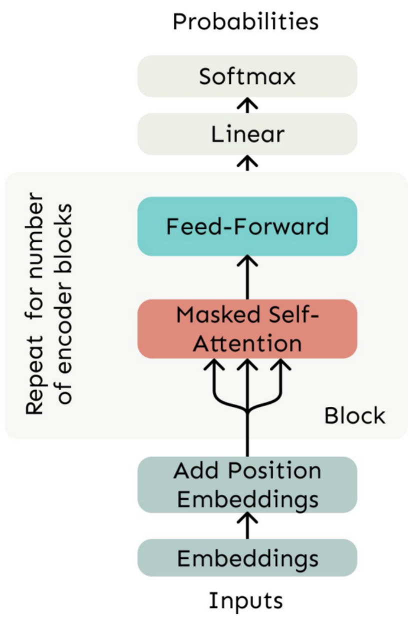


Figure 9: Transformer Decoder (Credit: John Hewitt)

Where an encoder block is one complete self-attention + feed-forward unit.

7.3 The Transformer

In 2017, *Attention Is All You Need* (Vaswani, et al.) was published describing the Transformer architecture. Since then, it has become one of the most cited papers of the 21st century. We will take it step by step to understand its importance.

7.3.1 Multi-Head Self Attention

Up until now, we've discussed the *self-attention* mechanism where we create a new vector for each word based on what looking at all previous words in the sequence. Single-head attention, what we've discussed so far, is great at letting every token look at every token and decide how important they are. For example, in the sentence:

```
The animal didn't cross because it was too tired.
```

The token "it" strongly attends to "animal" because their meanings are linked. Single-head attention is great at letting tokens build contextual understandings by mixing information from other positions. However, the operation:

$$\mathrm{softmax}(QK^\top)V$$

which we've been working with so far only produces a single weighted average of values V. That means the model is only focused on one pattern of relationships per layer. But language often has multiple simultaneous relationship! For example:

```
The stock rose because of earnings.
```

has relationships of subject-verb, modifier-noun, cause-effect, and time order. Our single-head attention won't be able to focus on all of these relationships at once.

Multi-head attention fixes this by giving the model multiple independent sets of Q, K, V projections. Each "head" learns to look for different kinds of relationships in the data. One head may focus on syntax, another on relationships, another on semantics, another on position, and so on and so forth. When all heads finish attending, we concatenate their results and mix them again linearly.

For an integer number of heads k, each an independent self-attention mechanism, we define matrices $K^\ell, V^\ell, Q^\ell \in \mathbb{R}^{d\times\frac{d}{k}}$ for ℓ in $\{1, \ldots, k\}$. These are our key, query, and value matrices for each head. This means we transform the same input into k different subspaces of dimension $\frac{d}{k}$. We then compute:

$$q_{1:n}^{(\ell)} = x_{1:n}Q^{(\ell)} \quad k_{1:n}^{(\ell)} = x_{1:n}K^{(\ell)} \quad v_{1:n}^{(\ell)} = x_{1:n}V^{(\ell)}$$

as in self-attention but with shape $\mathbb{R}^{n\times(\frac{d}{k})}$. For each head ℓ, compute the pairwise similarity between queries and keys:

$$s_{ij} = (q_i^{(\ell)})^\top k_j^{(\ell)}$$

Then normalize with a softmax over all j:

$$a_{ij}^{(\ell)} = \frac{\exp(s_{ij}^{(\ell)})}{\sum_{j=1}^{n} \exp(e_{ij}^{(\ell)})}$$

Then each head forms a new representation $h_i^{(\ell)}$ for every token:

$$h_i^{(\ell)} = \sum_{j}^{n} a_{ij}^{(\ell)} \mathbf{v}_j^{(\ell)}$$

which is the essence of attention - a weighted average of values. However, note that the output $h_i^{(\ell)}$ of each head is in dimension $\frac{d}{k}$. Thus, we concatenate all of the head outputs, letting $O \in \mathbb{R}^{d \times d}$:

$$h_i = O[h_i^1, \dots, h_i^k]$$

such each head output has dimensionality $d \times \frac{d}{k}$ and their concatenation will have dimension $d \times d$. This mixes information from all heads back into a single representation per token.

Now when we are computing a similarity score $s_{ij} = (q_i^{(\ell)})^\top k_j^{(\ell)}$, each score is a sum of d dimension dot products. Thus, as the embedding dimension d grows, the softmax function turns these scores into probabilities by exponentiating them. If any raw score gets too big, the largest score will dominate the rest of the probabilities and the model will only focus on one token. To fix this, we scale down the raw dot product by $\sqrt{d}$:

$$s_{ij}^{scaled} = \frac{(q_i^{(\ell)})^\top k_j^{(\ell)}}{\sqrt{d}}$$

This leads us to our final **multi-head self attention** formula:

$$\boxed{\begin{aligned}\text{MultiHead}(X) = \text{Concat}\Big(&\text{Attention}(q_{1:n}^{(1)}, k_{1:n}^{(1)}, v_{1:n}^{(1)}),\\ &\dots,\ \text{Attention}(q_{1:n}^{(k)}, k_{1:n}^{(k)}, v_{1:n}^{(k)})\Big) W_O\end{aligned}}$$

Where:

$$\text{Attention}(q_{1:n}, k_{1:n}, v_{1:n}) = \text{softmax}\left(\frac{q_{1:n} k_{1:n}^\top}{\sqrt{d}}\right) v_{1:n}$$

and W_O is the output projection that linearly mixes the concatenated attention back to a single d-dimensional representation per token.

7.3.2 The Transformer Decoder

Now that we've replaced self-attention with multi-head attention, we'll go through two optimization tricks which complete our transformer decoder. We loosely refer to these two together as "Add & Norm."

When we start stacking many of these layers on top of eachother, two problems arise:

- Information distortion $\rightarrow$ each attention layer completely transforms the representations. If you just keep feeding outputs into the next one, earlier information can be lost or over-written.
- Training instability $\rightarrow$ the deeper the stack, the harder it is to keep gradients well-scaled. Values start exploding or vanishing.

Residual Connections:
Residual connections are tricks to let the mode train better by simply adding the input of a layer to the output of that layer.

$$f_{residual}(h_{1:n}) = f(h_{1:n}) + h_{1:n}$$

This means that the sublayer only needs to learn *corrections* to the input, not the whole representation from scratch. It is much easier to learn the difference of a function from the identity function than it is to learn the function from scratch!

Layer Norm:
As we stack many layers, the scale and distribution of the activations (the output for each each token as a vector after multi-head attention) inside the network keep changing. Some tokens may have high values, some lower. This makes training unstable as gradients can explode or vanish, small numeric differences early in training get amplified, and learning rates become hard to tune. Layer normalization keeps every token vector "well-behaved" - with the goal of zero mean (μ) and unit variance (σ) per layer.

Just like that, we've finished the Transformer decoder!

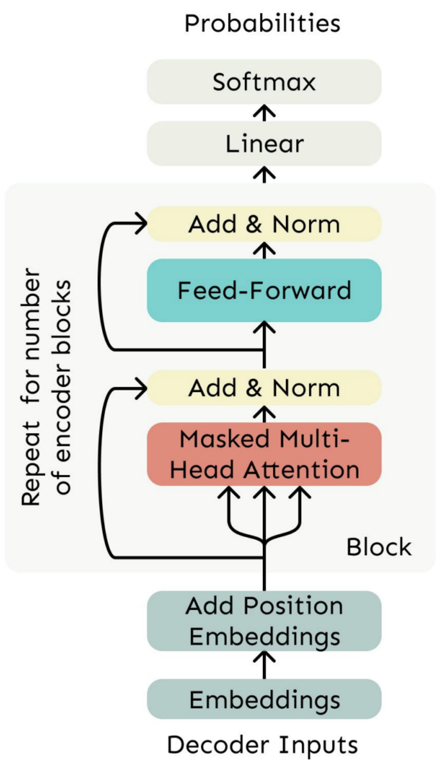


Figure 10: Transformer Decoder (Credit: John Hewitt)

Each block is the “processing unit” the model repeats over and over (for example, 6 times in the original Transformer, 12 in GPT-2 small, 96 in GPT-3). Each block consists of:

1. Multi-head attention
2. Add & Norm
3. Feed Forward
4. Add & Norm

Of course, the greater the amount of blocks the greater amount of meaning and nuances the model is able to capture.

7.3.3 The Transformer Encoder

The decoder we’ve studied so far looks only backward in a sequence (because of the masking - formally we call this unidirectional). However in tasks like translation, summarization, and classification we already have the full input sequence and we want to understand it *bidirectionally*. This is what the transformer encoder is for - building a contextual representation of the entire input sequence where each token can see all others. The only difference is we remove the masking!

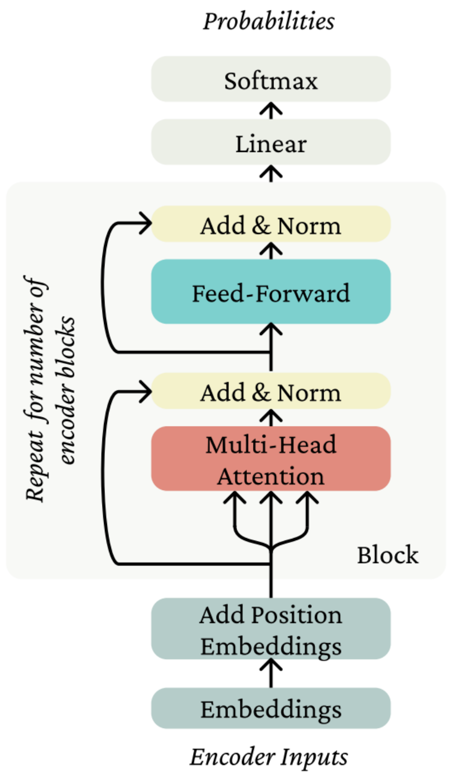


Figure 11: Transformer Encoder (Credit: John Hewitt)

7.3.4 The Transformer Encoder-Decoder

When we put the transformer encoder and decoder together, we can process the input sentence bidirectionally and generate the target unidirectionally. For example, in translation or in article summarization:

1. The encoder reads the source sequence and can fully understand the entire sentence's semantics since it can see entirely.
2. The decoder generates the target sequence by using masked multi head-attention over what is has generated so far and cross-attention over the

encoders output.

This setup — bidirectional encoder + unidirectional decoder — is occasionally called a sequence-to-sequence (seq2seq) transformer. Notice we have mentioned **cross-attention**, effectively it is just attention where queries come from the decoder and keys/values come from the encoder. Intuitively, each decoder token asks, “which encoder tokens are relevant to what I’m trying to generate next?”

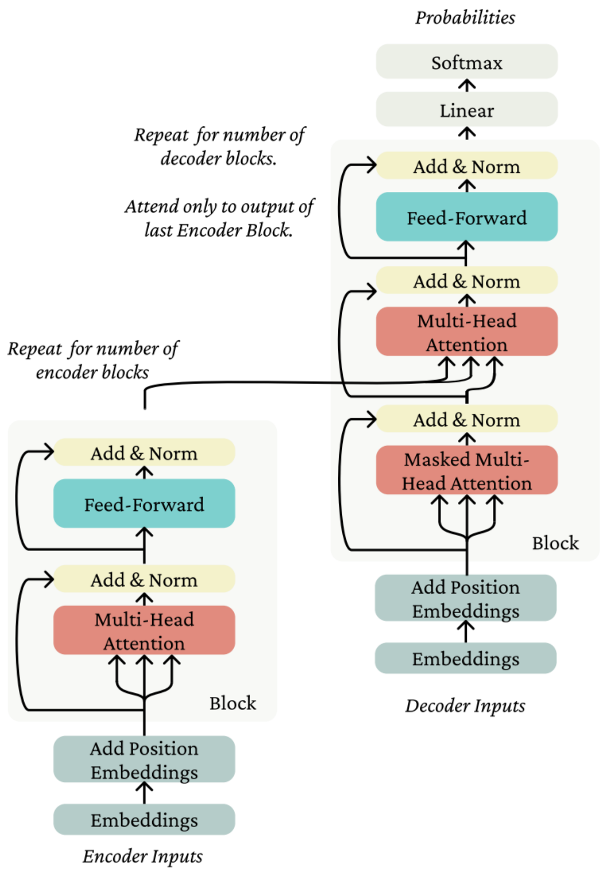


Figure 12: Transformer Encoder-Decoder (Credit: John Hewitt)

8 Pretraining

Pretraining is the initial phase in an LLM where it learns from a massive, diverse dataset of trillions of tokens with the goal of gaining a better understanding of context, semantics, language, and different types of knowledge. Pre-training is extremely computationally expensive and requires incredible amounts of data. In modern NLP, almost all parameters in NLP models are initialized in pre-training. In this note we understand the pre-training process and capacity more thoroughly.

Recall we have some learnable parameters θ in our distribution p_θ, a data distribution $\mathcal{D}$, and we're going to optimize:

$$\min_\theta \mathbb{E}_x \sim \mathcal{D}[-\log p_\theta(x)]$$

However, in practice we can't feed several thousand books as one long sequence into GPUs. Instead, we split text into chunks for the GPUs. By **padding**, we keep sentence boundaries and pad them with a special [PAD] token so they all have similar sizes. However, by doing so we need to filter out documents longer than n and we end up wasting our compute on useless blanks. Instead, we do **packing** - we string together a bunch of documents and process them as one continuous stream. Packing keeps GPUs busier and changes the effective data distribution to predicting each token conditional on all previous tokens in that packed stream. Thus, we change our notation as:

$$\min_\theta \mathbb{E}_x \sim \tilde{\mathcal{D}}[-\log p_\theta(x)]$$

Visually:

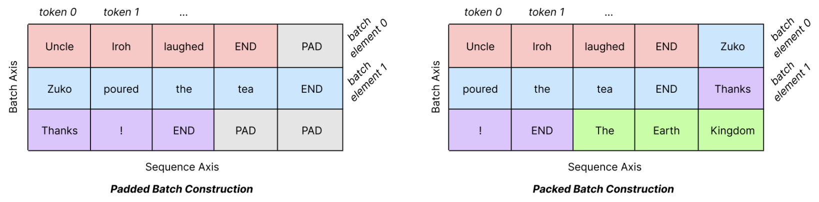


Figure 13: A comparison of padding and packing (Credit: John Hewitt)

8.1 Data Distribution

Next we want to understand how big exactly the data we're pretraining on is. In any language model, the training loss depends on three controllable quantities:

1. Model size - the number of parameters the model contains
2. Data size - number of tokens seen during training

3. Compute - how much optimization we can afford

However, there is a tradeoff:

- With too parameters and too much data → underfitting. The model can't represent all the patterns it encounters.
- Too many parameters and too little data → overfitting. The model is only memorizing training text.

There is an *optimal balance* between $N_{parameters}$ and N_{data} for a given compute budget. Solving the compute-optimal frontier is a large topic of research[3].

Lets consider the size of datasets used in practice for pretraining the GPT style models:

Model	**Provider**	**Year**	**# Parameters**	**Corpus Size (# Tokens)**
GPT-1	OpenAI	2018	117M	1.3B (BooksCorpus)
GPT-2	OpenAI	2019	1.5B	21B (WebText)
GPT-3	OpenAI	2020	175B	∼500B (CommonCrawl)
GPT-4	OpenAI	2023	—	—
GPT-5	OpenAI	2025	—	—
OLMo-2	AllenAI	2024	7–13B	5T
Gemma 3	Google	2024	270M–27B	6–12T
DeepSeek-V3	DeepSeek AI	2024	685B	14.8T
Llama 3	Meta	2024	7–405B	15T
Qwen 3	Alibaba	2025	235B–1T	36T

Table 3: Pretraining data size and model parameters for major large language models.

How difficult is it to get to this level of data distribution? *Really difficult.* Lets assume we start with the entire Internet, and then do some necessary filters:

1. Text extraction from HTML.
2. Language filtering for only English data.
3. URL filtering to avoid adult content.

[3]Consider a recent paper *Pre-training under infinite compute* published by Kim et al (2025). We are building faster computers ("infinite compute") but running out of high-quality text for them to read ("fixed data"). The authors found that the standard approach of just making models bigger backfires because they eventually just memorize the limited data (overfitting) rather than learning from it. To solve this, they propose a new recipe: first, they use significantly stronger "regularization" (penalties for complexity) to prevent overfitting, and second, they discover that training a "committee" of smaller models and averaging their answers (ensembling) actually performs better than training one giant model. This method is incredibly efficient, sometimes requiring 17 times less data to learn a subject like math. Crucially, they show you can "teach" a single small model quality outputs by distilling it, giving you high performance without needing a massive supercomputer to run it.

4. Deduplication.
5. Removing personally identifiable information (ex: SSNs)

This has been done before - DataCompLM removed nearly 99% of original documents that they scraped from the entire Internet. They still ended up with roughly 2-3 trillion tokens.

Model Based Comps
Another idea is the following: since we know that Wikipedia (around 5-7 billion tokens) is a very high quality dataset, why don't we score documents to how similar they are to Wikipedia. We train this classifier to prioritize (i.e., give a high score to) documents that look like or contain Wikipedia text and de prioritize (i.e. give a low score to) documents that don't look like Wikipedia text. We can then label each document with this classifier, sort our training pool by the associated score, and take the highest scoring documents as our pretraining data.

8.2 Summary

Pretraining is the building block of how the model actually learns. Think of it as the "first stage" of the model learning, where its learning basic features on the data and the parameters are being defined. However, we see that data is the primary bottleneck for high quality pretraining.

9 Post-training and fine-tuning

Now that we have our language model trained, how do we get it to do what we want it to do? This is the theory of post-training and fine-turning. A pre-trained model does not answer to questions - it simply models continuations of text. If we take a P_θ, and sample a sequence from it, what kind of sequence do we get? It should give us web documents, the exact data it was trained on! But this is not what we want! Our goal is to use P_θ as a *chatbot*. Let:

$$x = \text{Where is Columbia University located?}$$

From a human perspective, this is obviously a question. However, from a models perspective, this is just a string of texts. Our first attempt (response) is:

$$\hat{y} \sim P_\theta(y|x) \rightarrow \text{Where is NYU located?}$$

Why is this reasonable given the current model we have? In the training data, its not necessary the case the input is a Q&A. In fact, it is likely the input is something more general. The model we trained is to pick up patterns to generate sequences the list of patterns - thus this is a fair response. Our second attempt (response) is:

$$\hat{y} \sim P_\theta(y|x) \rightarrow \text{"I was wondering this as I arrived in Penn Station."}$$

It could be equally likely this is the output since narrative continuation is a somewhat likely continuation! To define the problem, our language model is **underspecified** - it does not know what we want. Formally, while we want $P(y|x, \text{ answer the question })$, we only have $P_\theta(y|x)$.

So how do we get the language model to do what we want? This is called **alignment**. The recurring issue is we don't want to tell it what to do constantly. This is especially difficult for generation systems that can handle a broad array of questions. The basic way to do alignment is to give the LM examples. We introduce **few-shot prompting**:

$$\{(x^{(1)}, y^{(1)}), \ldots, (x^{(i)}, y^{(i)})\}$$

Where:

$$x^{(1)} \rightarrow \text{How do you fix a kitchen sink?}$$

$$y^{(1)} \rightarrow \text{Okay, fixing a kitchen sink ...}$$

And:

$$x^{(2)} \rightarrow \text{What is the capital of France?}$$

$$y^{(2)} \rightarrow \text{Paris}$$

At inference time, we sample:

$$\hat{y} \sim P_\theta(y|(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(k)}, y^{(k)}), x)$$

What does the model learn from this process? For starters, it learns more prefix information. For example, the model can pick up that questions usually indicate a user-fed question, as opposed to a start of a story. This makes it easier to specify what you want the model to do. By specifying information in the prefix, we explain the sort of output we want. So we are not changing the model parameters, we are changing the conditioning context. This is why few-shot prompting works, it exploits the models ability to do in-context pattern inference.

But no models nowadays have this - why? It makes inference more expensive! Inference costs increase with prompt length. Since self-attention is roughly $O(T^2)$, where T = total number of tokens. Few shot prompting increases T as it now has a longer context, more memory bandwidth, and higher dollar per query! For systems generating millions of tokens per sample, this matters.

What we do instead is take the data we trained the model with and **finetune** it. Instead of specifying behavior at inference time, we want the model to internalize it. How do we do this? The key is to constantly update our parameters θ. We construct a supervised dataset:

$$D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(i)}, y^{(i)})\}$$

where $x^{(i)}$ is an instruction or question and $y^{(i)}$ is the desired response. Starting from pretrained parameters θ_P, we minimize the negative log-likelihood:

$$\mathcal{L}(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{D}}\big[-\log P_\theta(y \mid x)\big],$$

using stochastic gradient descent:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta).$$

However, fine-tuning is odd because if we succeeded in taking the min, the initial θ_P should not matter. Although the optimization objective is the same, the initialization θ_P is crucial.

- **Optimization depends on the starting point:** Neural network training is highly non-convex, meaning gradient descent does not find a single global optimum. Where the model ends up depends strongly on where it starts. Initializing at θ_P places the model in a region of parameter space that already produces coherent language, making fine-tuning efficient and stable.

- **Language structure is already learned:** Pretraining teaches the model the basic structure of language, including grammar, meaning, and common patterns of text. Fine-tuning does not need to relearn these fundamentals; it only adjusts how the model responds to certain inputs. Starting from θ_P allows fine-tuning to build on existing knowledge rather than learning from scratch.

Effectively, during fine-tuning we do not want the parameters to move too far from the pretrained solution θ_P. To make this notion precise, we treat all model parameters as a single vector:

$$\theta \in \mathbb{R}^G,$$

where G is the total number of parameters (e.g. embeddings, attention weights, and feed-forward layers). We then measure the distance from the pretrained model using the squared ℓ_2 norm:

$$\|\theta - \theta_P\|_2^2.$$

This leads to an ℓ_2-regularized fine-tuning objective:

$$\min_\theta \; \mathbb{E}_{(x,y)\sim\mathcal{D}}\big[-\log P_\theta(y \mid x)\big] \;+\; \beta\|\theta - \theta_P\|_2^2,$$

where:

- $\mathbb{E}[-\log P_\theta(y \mid x)]$ encourages the model to fit the fine-tuning data,
- $\beta > 0$ controls the strength of the regularization,
- $\|\theta - \theta_P\|_2^2$ penalizes large deviations from the pretrained parameters.

This objective makes the role of initialization explicit. Fine-tuning is no longer just about minimizing prediction error on the new dataset, but about finding parameters that both perform well on the task and remain close to the pretrained model. In this sense, θ_P acts as a strong prior, ensuring that alignment modifies behavior without destroying the linguistic structure learned during pretraining.

Regularizing by $\|\theta - \theta_P\|_2^2$ measures distance in *parameter space*. But what we ultimately care about is not whether the *weights* changed—it is whether the *model's behavior* changed. A more direct way to measure behavioral change is to measure distance between the model's output *distributions* in probability space. Concretely, for a given context (prefix) $x_{<t}$, the model defines a next-token distribution over the vocabulary V:

$$P_\theta(\cdot \mid x_{<t}) \in \Delta^{|V|-1}.$$

We can therefore penalize the model for moving too far from the pretrained distribution $P_{\theta_P}(\cdot \mid x_{<t})$ using the **KL divergence**:

$$D_{\mathrm{KL}}(P_\theta(\cdot \mid x_{<t}) \,\|\, P_{\theta_P}(\cdot \mid x_{<t})) = \sum_{w\in V} P_\theta(w \mid x_{<t}) \log \frac{P_\theta(w \mid x_{<t})}{P_{\theta_P}(w \mid x_{<t})}.$$

Intuitively, this term is small when the fine-tuned model assigns probability mass to tokens similarly to the pretrained model, and it grows when the fine-tuned model shifts probability toward different tokens.

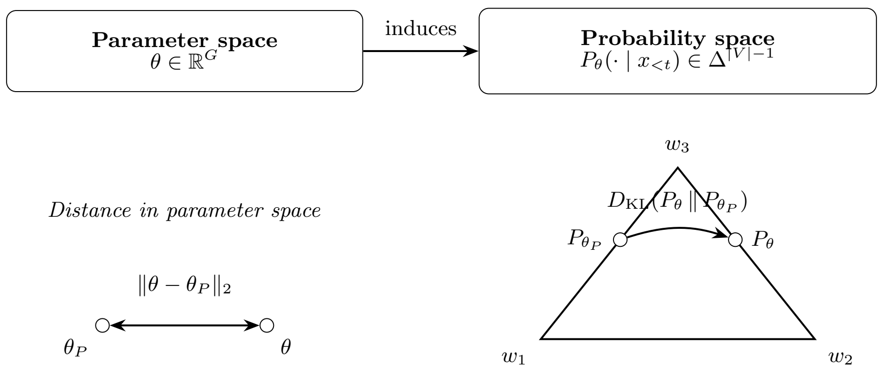

Parameter space
$\theta \in \mathbb{R}^G$
induces
Probability space
$P_\theta(\cdot \mid x_{<t}) \in \Delta^{|V|-1}$
Distance in parameter space
$\|\theta - \theta_P\|_2$
θ_P
θ
w_3
$D_{\mathrm{KL}}(P_\theta \,\|\, P_{\theta_P})$
P_{θ_P}
P_θ
w_1
w_2

Figure 14: Two ways to measure "how far" a fine-tuned model moved from pretraining: distance in parameter space versus divergence between next-token distributions for a fixed context $x_{<t}$.

Thus, KL regularization constrains fine-tuning in *probability space*: it discourages large changes in next-token predictions even if many different parameter settings could produce them.

Prefix Tuning

Few-shot prompting controls model behavior by prepending a textual prefix to the input at inference time. Prefix tuning replaces this fixed, hand-written prefix with a *learnable prefix* trained end-to-end. Instead of modifying the model parameters, prefix tuning introduces a small number of virtual tokens whose embeddings are optimized while the base model remains frozen. These virtual tokens are prepended to the input sequence and participate in the attention mechanism just like real tokens. Concretely, we introduce k *virtual tokens* represented by a learnable matrix

$$P \in \mathbb{R}^{k \times d},$$

where d is the model's embedding dimension. For an input sequence x with token embeddings

$$X \in \mathbb{R}^{T \times d},$$

we form an augmented sequence by concatenation:

$$X' = \begin{bmatrix} P \\ X \end{bmatrix} \in \mathbb{R}^{(k+T) \times d}.$$

The model then runs exactly as usual on X', and we train only the prefix parameters P (keeping the pretrained weights θ fixed) by minimizing the negative log-likelihood:

$$\min_P \; \mathbb{E}_{(x,y)\sim\mathcal{D}}\big[-\log P_\theta(y \mid [P; x])\big].$$

Intuitively, the prefix vectors behave like an optimized prompt: they participate in attention like real tokens and steer generation, but they are learned directly from data. Since we are only additionally training on this smaller input, this is much more efficient.

LoRA
LoRA is another fine-tuning method, but instead of adding learnable input tokens, it adapts the model by learning small, structured updates to selected weight matrices inside the network. Let $W \in \mathbb{R}^{d\times d}$ be a pretrained weight matrix (for example, one of the linear projections in self-attention). In LoRA, we replace W with

$$\tilde{W} = W + \Delta W,$$

where the update ΔW is constrained to be low-rank:

$$\Delta W = AB^\top, \qquad A \in \mathbb{R}^{d\times r},\ B \in \mathbb{R}^{d\times r}, \qquad r \ll d.$$

During training, the original matrix W remains frozen and only the low-rank factors A and B are updated. This reduces the number of trainable parameters from d^2 to $2dr$ per modified matrix, which can be a dramatic savings when r is small. The motivating assumption is that task-specific changes to a pretrained model often lie in a low-dimensional subspace. A low-rank update captures the most important directions of change while preserving most of the pretrained representations.

9.1 Supervised fine-tuning

Now that we have fine-tuned the model while its in use, how can we make the model better over time? Recall we have the goal of taking P_θ and getting a variety of outputs, each high quality. Recall,

```
Columbia University is located in _____.
```

There will be a monotonically decreasing probability distribution of the highest likelihood words.

If we sample from $P(\cdot|W_{<t})$, the sum of probabilities over individually lower-probability words will be high. For example, the sum of $2:\epsilon$ can be 10%. The problem is, failing 10% is bad! The intuition for **Top-p nucleus sampling** is that the lowest probability words in each context is probability bad. Top-p sampling is a rather simple algorithm:

- Pick a percentage p (ie. 90%)
- Sort the vocab from highest-to-lowest probability for any output
- Keep words that compose that 90%
- Set the rest to 0
- Re-normalize and distribute the remaining probabilities.

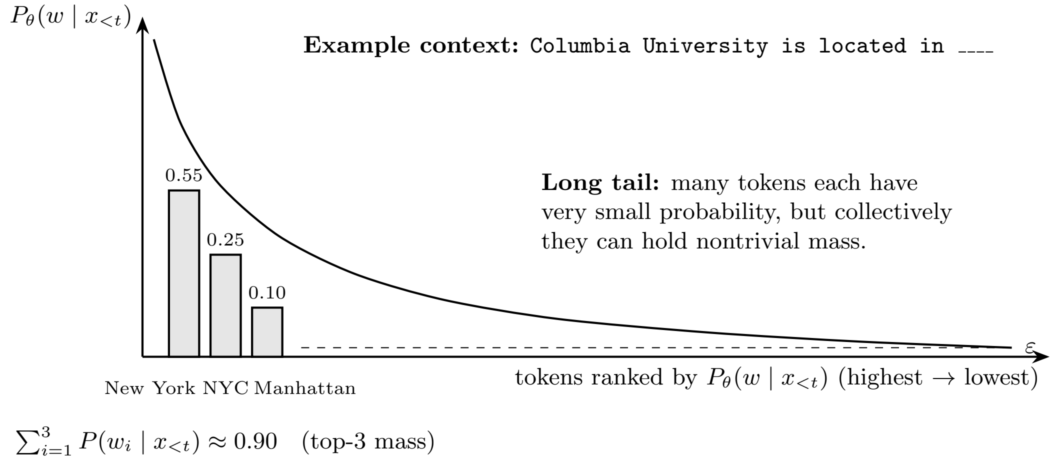


Figure 15: A typical next-token distribution exhibits a heavy head and a long tail. In this example, the top three tokens already account for approximately 90% of the probability mass, while the remaining vocabulary forms a long tail approaching ε. This motivates top-p (nucleus) sampling: retain only the smallest set of tokens whose cumulative probability exceeds p, then renormalize.

The algorithm is so simple, formalizing it is unneccesary. This makes a *huge* difference in generation since circa. 2019. Nowadays, we do $P = .99$ which is really close to the original sampling distribution[4]

Preference Learning

But what if we don't know our desired output y for a given input x? Then, we want the model to learn from preferences. If I'm given $\hat{y}_1, \hat{y}_2$, I can decide which I like better.

Lets say I prefer $\hat{y}_1 > \hat{y}_2$, the model will be pushed towards my favorite answer. A simple algorithm for this is **direct-reference optimization**. Let

$$\mathcal{D} == \{(x, y_c, y_r\}$$

Where y_c is chosen and y_r is rejected. Then, DPO-loss is:

$$\mathcal{L}_{\text{DPO}}(\theta) = -\,\mathbb{E}_{(x,y^+,y^-)\sim\mathcal{D}}\left[\log\sigma\left(\beta\left(\log\frac{P_\theta(y_c \mid x)}{P_{\theta P}\text{ref}(y_c \mid x)} - \log\frac{P_\theta(y_r \mid x)}{P_{\theta P}(y_r \mid x)}\right)\right)\right],$$

where $\sigma(\cdot)$ is the logistic sigmoid function and $\beta > 0$ controls the strength of the preference optimization. Intuitively, DPO increases the relative likelihood

[4]Consider also *top-k sampling*. Instead of the model picking from the entire vocabulary it samples from the top k choices (e.g. top 10) then samples one randomly. This prevents nonsensical outputs while keeping some degree of variation.

of preferred responses while decreasing the likelihood of dispreferred ones, measured against a fixed reference policy. The reference policy acts as an implicit KL regularizer, ensuring that optimization shifts behavior toward human preferences without drifting too far from the original model.

However, DPO assumes that preference pairs (x, y_c, y_r) are drawn from distributions that are reasonably close to the current policy. In practice, however, preference data is often collected *off-policy*: the responses y_c and y_r may have been generated by older models, different systems, or even humans. As a result, their distribution may differ significantly from the current policy P_θ. To handle this setting, we can instead learn an explicit *reward function* that assigns a scalar score to responses. Given a dataset of preference comparisons

$$\mathcal{D} = \{(x, y_c, y_r)\},$$

we train a reward model $r_\phi(x, y)$ such that preferred responses receive higher scores than dispreferred ones.

9.2 Summary

In summary, all methods are parameter-efficient ways to adapt a pretrained model:

- **Few-shot prompting** prepends examples of successful prompts at inference time so the model can see what a successful user query can be. However, this is inefficient due to adding additional token context.
- **KL-divergence** is used during post-training to regularize fine-tuning by penalizing deviations from a reference model, ensuring that the learned policy remains close to the original distribution while incorporating preference or reward signals.
- **Prefix tuning** learns additional *input-side* parameters (virtual tokens) that steer attention and generation.
- **LoRA** learns low-rank *weight updates* to selected internal matrices, while leaving the original weights unchanged.

In all cases, the goal is the same: achieve task-specific behavior with minimal parameter updates and minimal disruption to the pretrained model. We then improve the model over time using:

- **Top-p sampling** where we keep the output words that compose the p percent highest likelihood.
- **Top-k sampling** where we keep the output words that compose the k most likely output words.
- **Direct reference optimization** where we train another model to have positive reference responses, giving the model human preference optimization and shifts.