The Esoteric Programming Language at the Heart of Modern AI

The Art of Transformer Programming

Human Powered Optimization Procedures

PREPRINT

Yaniv Leviathan

The Art of Transformer Programming

Yaniv Leviathan

Version 2022.10.24-2023.10.30

To my incredible wife and her unwavering support,
and to my amazing mom, who taught me everything that I know.

Yaniv Leviathan

Science is what we understand well enough to explain to a computer. Art is everything else we do.

Donald E. Knuth

About This Work

What I cannot create, I do not understand.

RICHARD FEYNMAN

"How can modern large language models perform simple computations, like sorting a list, searching for a sequence, or adding two numbers in decimal representation?"

Surprisingly, several years and thousands of papers since the Transformer was invented, it is still hard to find a satisfying answer online to this seemingly simple question. Being a firm believer in Richard's Feynman's words above, I decided to answer this question myself.

In fact, I decided to choose a set of simple programs, beyond those above, and implement all of them *by hand* on top of a production grade decoder-only Transformer. These programs included the canonical printing of the fixed string "Hello World!", looking up values in a fixed memorized lookup-table, searching for an input pattern in a given longer sequence, sorting a list of numbers, and of course, decimal addition. Luckily, with some free holiday time, a few days later I found myself with several Python colabs with implementations of these programs. While my curiosity was appeased, with the hope that others might find the work interesting I decided to dedicate some free weekend time since and compile the work into a short booklet. This manuscript is the result.

I found this problem interesting for two reasons. First, creating these Transformer programs by hand was an extremely fun set of puzzles. The intellectual challenge felt similar to trying to program with an esoteric programming language, like Befunge, BCL, Brainfuck, Chef, INTERCAL, Malbolge, Piet, and others. Programming a Transformer manually, like is done in this work, *is* an esoteric programming language, made mostly of a particular set of linear algebra operations. As a matter of fact, if you feel especially bored, Chapter 8 includes

a simple interpreter for this esoteric language... but notice that most modern AI systems are such interpreters themselves! Which brings me to the the second, and more important, reason.

The second reason that this problem interested me is that this said esoteric programming language also happens to power one of the most popular "computers" in the world - namely the Transformer architecture behind modern AI systems. As of today, Transformers are amongst the most important backbones of AI systems, and understanding better how they work, and especially, what is *hard* for them, is key to improving them. In spite of a tremendous amount of AI research since its invention, the Transformer architecture proved to be extremely robust, with very few changes being adopted in practice. We'll soon see that while many of the design choices made by the Transformer creators make things harder for our human brains, these don't pose any issue to the optimization procedure and the Transformer architecture. The most interesting cases though, are those that are hard for us humans *and* are hard for the optimizer or the architecture, and understanding these better might be key to creating better AI systems.

A Note About the Puzzles

> *A good puzzle, it's a fair thing. Nobody is lying. It's very clear, and the problem depends just on you.*
>
> ERNO RUBIK

> *Once I get on a puzzle, I can't get off.*
>
> RICHARD FEYNMAN

I hope that reading this booklet can help others get a better grasp of how Transformers work. That said, and to throw another quote into the mix (this time by Richard Branson): "The best way of learning about anything is by doing". To that end, I included a small set of puzzles as exercises throughout the book, at the end of each of the chapters. To help you orient yourself, and more easily decide which puzzles to attempt, I added some special markings:

Hard All especially hard puzzles are marked with the black chess queen symbol (♛). These can usually take an hour or two and sometimes even more. If a puzzle is not super hard, but does require writing a lot of code I used this symbol as well. Rarely, these puzzles can be solved very quickly, but do require some "aha moment" or special

insight.

Easy Those puzzles that are especially easy, I marked with the white pawn (♙) symbol. These should all take less than a minute or two to solve, and sometimes mere seconds. Almost all of these can be solved "in your head", and don't even require a pen or paper, and so there is no reason to skip them. If I used this mark for an exercise that requires writing code, that means that either it's extremely trivial code, or that you should be able to imagine it mentally. These time estimates assume that you are fully familiar with all of the terms and concepts in the puzzle - they might take much longer if you are not.

Recommended Regardless of difficulty, the puzzles that are my very favorite and I recommend the most are marked with the musical note symbol (♪).

Unsolved by me There are very few puzzles that I have not yet solved myself. Those are marked with my initial (𝒴). If you managed to solve them - huge congrats! If your solution also happens to be correct, I'd really love to hear about it!

Open-ended I usually much prefer rigorous puzzles, those that you can *solve*. I did include some more open-ended puzzles, mostly to encourage generally thinking about a topic. I did mark all of these with the open-ended symbol ('…'), so you can skip them if you prefer only the rigorous ones.

While there are some correlations between the symbols, I used the marks independently, so some puzzles brand more than one.

Skippable Finally, one more symbol that applies to sections and not puzzles, is the skippable symbol (↯). I used it to denote sections from the main text that are completely optional and may be safely skipped.

The following table summarizes the notations:

Symbol	Meaning	Symbol	Meaning
♛	Extra hard.	𝒴	Not yet solved by me.
♙	Extra easy.	'…'	Open ended.
♪	One of my favorites.	↯	Safely skippable.

If there is interest, I might include my solutions to the puzzles within the book in a future update.

In addition to the puzzles, if you have the time, I strongly recommend treating the main

constructions in the book as puzzles as well - reading only the problem definitions at first and trying to implement the Transformer programs *yourself*. It should not take more than a few hours per program (and often, much less) and there is no better way to learn. If you'd like to attempt this yourself, note that most of the programs can be implemented in no more than 50 lines of readable code (most can be implemented in much less). I added spoiler warnings after each of the formal problem definitions in each of the chapters, so that you can read without risk of accidental spoilers even in case you'd like to attempt solving everything yourself before reading. If you do decide to implement any of the programs in this work, and your results are interesting (e.g. more efficient than mine), I'd really love to hear about it.

More generally, I'd really love to hear of any comments or suggestions for improvement you might have, and especially of any errors you find, so please reach out!

Note About the Code Snippets

Beware of bugs in the above code; I have only proved it correct, not tried it.

DONALD E. KNUTH

Programs must be written for people to read, and only incidentally for machines to execute.

HAROLD ABELSON

Often reading a short piece of code is many folds easier to understand than reading an equivalently lengthy math formula. With that in mind, code snippets are included throughout this work. They aren't very practical and you should feel free to treat them exactly as "easier to understand formulas". Specifically - *this booklet can be read in bed!*

That said, if you choose to dig deeper (and do the exercises) the code should all be runnable. Note though, that the snippets are meant to be run in order, and might depend on preceding snippets. You can find all code snippets in the book in colab form at: https://yanivle.github.io/taotp.ipynb.

Thank you!

Yaniv Leviathan

I have made this longer than usual because I have not had time to make it shorter.

BLAISE PASCAL

Introduction

Everybody should learn to program a computer, because it teaches you how to think.

STEVE JOBS

THE Transformer [Vaswani et al., 2017] is a highly efficient *differentiable computer*. When equipped with the right set of weights, obtained through a lengthy optimization process on supercomputers processing massive amounts of data, Transformer models are able to recall vast amounts of memorized knowledge and perform complex computations. Unlike human-designed programs on traditional computers, the inner workings of such models after training are not well understood. Many studies aim at analysing the programs obtained by these optimisation processes, with some, albeit limited, success. In this work we ask, and attempt to answer, a simpler question - how *could* a set of simple programs be efficiently implemented on a Transformer computer?

An entire subfield of AI, namely *Mechanistic Interpretability*, is concerned with reverse engineering neural programs obtained from optimization processes (most recently with reverse engineering Transformer programs specifically) but surprisingly little effort (to my knowledge!) is spent on actually programming these manually. Imagine an alternate universe, where in 1940, aliens came to earth and supplied us with modern x86-based computers, as well as a magical device that could produce working program *binaries* from simple descriptions. Ask this magical device to produce a "photo editor" and the device would output the binary for a Photoshop-like program, ask it for a "first person shooter" and get the binary for Doom, etc. The device would also get as input the maximum desired size of the binary - if you ask for a shooter with a binary consisting of only 1 KB, you'd get a degenerate version of Doom (if the size you request is too small, you won't get a working

program). In this alternate universe, you might expect that 1940s scientists would reverse engineer and analyze the binaries outputted by this magic device, maybe trying to restrict the size limit to make the generated programs simpler and analysis easier. This is analogous to mechanistic interpretability. You might also expect that in this alternate reality some scientists would try to learn x86 assembly code and actually *manually* write some programs, without using the magic device. The analogy here, is this book.

The question of how a program *could be* implemented on a Transformer is indeed different from asking how the program *is* implemented in a specific trained LLM. The latter might be more important for many practical applications, but the former is hopefully much easier to answer. For example, there are countless degrees of freedom provided by the Transformer architecture, like rotating the activations basically anywhere, that enable trivial entanglement of unrelated concepts (see Exercise 8 in Chapter 1). Making sense of how a specific trained model performs a calculation might require understanding and disentangling many of these. This distinction between the questions is somewhat similar to the distinction between asking how a biological brain *might* be able to perform a computation efficiently, and asking how our very specific human brains have happened to evolve to solve that calculation*.

Putting aside the ability of a Transformer to be efficiently optimized, we instead focus just on the Transformer as a programmable computer. It is known that variants of the Transformer are Turing-complete, but a repository of efficient implementations of simple programs on a Transformer computer is not widely available. Moreover, much of the implementations that do exist use simplifications compared to common production settings (e.g. outputting intermediate auxiliary tokens, using Transformer encoders, or ignoring components such as layer normalizations), which make the problems substantially easier. In this work, we will implement by hand a set of basic programs on a Transformer computer, i.e. we will manually set the weights of a Transformer, without a training procedure or datasets. We'll use a non-simplified decoder-only stack, similar to that used by well known large language models.

The programs we will implement include printing Hello World† (Chapter 2), looking up values in a Lookup Table (Chapter 3), Searching for patterns (Chapter 4), finding the Minimum and Maximum elements and Sorting a sequence of numbers (Chapter 5), and Decimal Addition (Chapter 6). While we'll tackle the programs mostly in increasing order of difficulty, we'll gradually develop a set of tools that we'll reuse from program to program. As we'll see, a large part of these tools will aim to overcome the mechanisms designed to aid the automatic optimization process. As a matter of fact, we'll observe that

* I'm personally much more interested in the former.

† We will use Small Caps to denote the names of our programs.

some of our implementations are unlikely to be achieved by an optimisation process. We will dedicate a significant part of the work to trying to make our programs efficient - i.e. use few parameters, and we'll then observe that with large enough Transformers, many of the implementation intricacies disappear.

The structure of the work is as follows: in Chapter 1 we will examine the implementation of the Transformer that we'll use. Then Chapters 2 through 6 will detail the construction of each of the programs we will implement. These can be partially read out of order, but do note that most of the later constructions leverage blocks developed in earlier sections. Finally, Chapters 7 and 8 are super optional, the former containing further thoughts about the sinusoidal embeddings and the latter offering a (somewhat idiotic) textual interpreter for Transformer programs.

A primary objective of this study is to increase the accessibility of the understanding of a Transformer computer. With that in mind, we'll prefer using simple Python code instead of lengthy equations where appropriate, include the majority of the code inline, and note that it can (and was) all run on a consumer grade laptop. We note that many of the constructions include a lot of technicalities, and while we try to make things as clear and self contained as possible, this book is not a good introduction to Transformers. Indeed, a solid understanding of the Transformer and its sub-modules, notably attention, MLP, and layer normalization, are recommended prerequisites for this text (resources such as [Alammar, 2018] might be helpful).

Acknowledgements

It is one of the blessings of old friends that you can afford to be stupid with them.

RALPH WALDO EMERSON

I'd like to extend a huge thanks to **Matan Kalman** and **Asaf Aharoni**, for reviewing the book and providing countless great thoughts, insightful suggestions, and encouragement. Another special thanks goes to **Danny Lumen** and **Dani Valevski** for super helpful comments. A big thanks also goes to everyone who supported this book and provided great words of encouragement: **Eyal Molad**, **Eyal Segalis**, **Yoav Tzur**, **Raya Leviathan**, **Liron Raz**, **Shlomi Fruchter**, **Andrei Broder**, **Avinatan Hassidim**, **Michael Daniel Birnberg**, and **Yossi Matias**. Thanks all for the great feedback and support! Another thank you goes to **Andrej Karpathy**, whose minGPT project[‡] and series of YouTube talks[§] were an inspiration for this work, and can act as great background material. A particularly special thanks goes to **Donald E. Knuth**, whose "The Art of Computer Programming" series ([Knuth, 1997a], [Knuth, 1997b], [Knuth, 1998]) is not only amongst my all time favorite set of books (the first three volumes of which I read back-to-back about 25 years ago!) but were the most direct inspiration for this booklet (including being its namesake). Oh, and he also developed TEX, without which this book would not have been possible. Big thanks to all the great minds whose inspiring quotes hopefully make this book more fun to read. And finally, a big thanks, as always, goes to my family, that put up with dad spending a few weekends writing this instead of doing shenanigans together.

‡ https://github.com/karpathy/minGPT

§ https://www.youtube.com/playlist?list=PLAqhIrjkxbuWI23v9cThsA9GvCAUhRvKZ

Contents

Part I

Preliminaries

Chapter 1

The Transformer

The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.

EDSGER W. DIJKSTRA

Computer science is no more about computers than astronomy is about telescopes.

EDSGER W. DIJKSTRA

1.1 Implementation

While the programs we'll develop are toy examples, our goal is to better understand real-world models. With that in mind, we'll use a standard implementation of a Transformer stack, following the original implementation from [Vaswani et al., 2017]. More precisely, we'll match our implementation of a standard Transformer during *inference*. As we'll see, it will often be easier to implement our programs on a Transformer encoder, but, as following [Radford et al., 2018] most modern large language models are decoder-only

Transformers, we'll use a decoder-only stack as well. Also common in modern large language models, following [Xiong et al., 2020], we'll use Pre-LN instead of Post-LN*. We will not use the sinusoidal encodings of the original Transfromer [Vaswani et al., 2017], and instead treat the positional encodings as "learned" embeddings, i.e. we will set them by hand, and count them towards our parameter counts (see Chapter 7). Finally, since dropout has no effect during inference (except possibly multiplying the weights by a constant), we will leave dropout out of our implementation entirely, without sacrificing the faithfulness of our implementation to the standard model.

See the full implementation in plain Numpy of a Transformer decoder that we'll use in Figure 1.1.

Since it is sometimes done in Transformer implementations (it was in the original Transformer paper), to potentially improve efficiency, and also just to increase the challenge, we will often follow [Press and Wolf, 2017], and tie the weights of the token embedding and output embedding matrices `tok_emb` and `out_emb` (in which case, we will only count one of them towards our parameter counts). Do note though that the motivations there do not hold for most of our programs (see Section 3.5). We leave those as separate arguments in our implementation for the cases when we chose to untie them (e.g. Section 3.5).

1.1.1 Inputs and Outputs

As we're using a decoder-only stack, the output of our programs will directly follow the input. Specifically, we'll refer to the contents of the block before we run our Transformer as the *input* and to anything added to the block by running our Transformer as the *output*. We will use *greedy decoding* for all of our programs, as implemented in Figure 1.2. Some programs will run for a fixed number of steps and some will optionally halt upon encountering the *end of sequence* (`<eos>`) token. Some programs will make use of a *beginning of sequence* (`<bos>`) sentinel token.

In most of the programs there will be a natural tokenization, and we will not use any fancy tokenizer. The main exception is the Hello World program (Chapter 2) where we will consider a problem-specific tokenizer as well as a more general tokenizer.

1.2 Transformer Programming

Each of our Transformer programs will simply consist of an assignment of values to all parameter arrays from Figure 1.1: `tok_emb`, `pos_emb`, and `out_emb` (as noted above we will

* We note that we have actually implemented all of the programs both with a Pre-LN and Post-LN setting, and, interestingly, Pre-LN turned out easier almost across the board, even for a human.

```python
import numpy as np

def norm(x, axis=-1, epsilon: float = 1e-10):
  return x / (x.sum(axis=axis, keepdims=True) + epsilon)

def softmax(x, axis=-1):
  return norm(np.exp(x - np.max(x, axis=axis, keepdims=True)), axis=axis)

def layer_norm(x, gamma = 1., beta = 0., axis = -1, epsilon = 1e-10):
  return beta + gamma * (x - x.mean(axis=axis, keepdims=True)) / (
    x.var(axis=axis, keepdims=True) ** 0.5 + epsilon)

def self_attn(x, Q, K, V, P, mask):
  queries, keys, values = (np.einsum('nj,hjk->nhk', x, M) for M in (Q, K, V))
  qk = np.einsum('nhk,mhk->nmh', queries, keys) / (Q.shape[-1] ** 0.5)
  attn_weights = softmax(np.where(mask[..., None], qk, float('-inf')), axis=1)
  x = np.einsum('nmh,mhk->nhk', attn_weights, values)
  return np.einsum('nhk,hdk->nd', x, P)

def transformer_layer(x, Q, K, V, P, M1, b1, M2, b2, ln1, ln2, mask):
  x = x + self_attn(layer_norm(x, **ln1), Q, K, V, P, mask)
  return x + np.maximum(layer_norm(x, **ln2) @ M1 + b1, 0) @ M2 + b2

def transformer(tokens, tok_emb, pos_emb, out_emb, layers, lnf):
  n, = tokens.shape
  causal_mask = np.tril(np.ones((n, n)))
  x = tok_emb[tokens] + pos_emb[np.arange(n)]
  for layer in layers:
    x = transformer_layer(x, **layer, mask=causal_mask)
  return np.einsum('nd,vd->nv', layer_norm(x, **lnf), out_emb)
```

Figure 1.1: The Transformer decoder implementation that we will use throughout this work. Remarkably, it comfortably fits in 30 lines of readable Numpy code.

```python
def decode(tokens, params, max_steps=1000, eos=None):
  for _ in range(max_steps):
    next_tok = np.argmax(transformer(tokens, **params)[-1])
    tokens = np.append(tokens, next_tok)
    if next_tok == eos: break
  return tokens
```

Figure 1.2: The naive implementation of greedy decoding that we'll use.

```
def base_layer(n_heads, d_model, d_ff, d_head):
    return {
        'Q': np.zeros((n_heads, d_model, d_head)),
        'K': np.zeros((n_heads, d_model, d_head)),
        'V': np.zeros((n_heads, d_model, d_head)),
        'P': np.zeros((n_heads, d_model, d_head)),
        'M1': np.zeros((d_model, d_ff)),
        'b1': np.zeros((d_ff,)),
        'M2': np.zeros((d_ff, d_model)),
        'b2': np.zeros((d_model,)),
        'ln1': {'gamma': 1., 'beta': 0.},
        'ln2': {'gamma': 1., 'beta': 0.},
    }

def base_params(vocab_size, block_size, d_model):
    return {
        'tok_emb': np.zeros((vocab_size, d_model)),
        'out_emb': np.zeros((vocab_size, d_model)),
        'pos_emb': np.zeros((block_size, d_model)),
        'layers': [],
        'lnf': {'gamma': 1., 'beta': 0.},
    }
```

Figure 1.3: Resetting the parameters of our Transformer. Note that due to the residual connections, `base_layer` initializes a Transformer layer to perform the identity function.

usually set `out_emb` to a copy of `tok_emb`), as well as `Q`, `K`, `V`, `P`, `M1`, `b1`, `M2`, and `b2`, for each of the layers. We include the `gamma` and `beta` parameters of all of the layer normalization modules in the implementation for completeness, but we will always use the default values of `gamma = 1` and `beta = 0`. Figure 1.3 illustrates setting of all the parameters. Note that we usually won't follow the popular conventions of having $d_{ff} = 4 \cdot d_{model}$ and $n_{heads} \cdot d_{head} = d_{model}$.

1.3 Efficiency

> *Make everything as simple as possible, but not simpler.*
>
> ALBERT EINSTEIN

We will aim to create programs that are *efficient*. More formally, we'll measure the efficiency of our programs by the number of their parameters. As we will see, sometimes we will need

```python
from functools import partial

class HumanReadableDict(dict):
    def __repr__(self) -> str:
        return "{" + ", ".join(f"{k}: {v:,}" for k, v in self.items()) + "}"

def count_params(params, count=lambda x: x.size):
    embs = ["tok_emb", "pos_emb"]
    if not np.all(params["tok_emb"] == params["out_emb"]):
        embs.append("out_emb")
    emb_params = {emb: count(params[emb]) for emb in embs}
    non_emb_params = 0
    for layer in params["layers"]:
        for param in layer.values():
            if isinstance(param, np.ndarray):
                non_emb_params += count(param)
    return HumanReadableDict(
        {"n_params": sum(emb_params.values()) + non_emb_params}
        | emb_params
        | {"non_emb": non_emb_params})

count_non0_params = partial(count_params, count=lambda x: (x != 0).sum())
```

Figure 1.4: Simple logic for counting the number of parameters and non-zero parameters of our programs. If `tok_emb` and `out_emb` are tied, we only count them once.

to make use of higher dimensions, e.g. for interim calculations, while most of the parameters will be zeroed, so as a secondary efficiency metric, we will also consider *sparse efficiency*, i.e. we will count just the number of parameters that are non-zero. Figure 1.4 implements the `count_params` and `count_non0_params` functions for that purpose.

1.4 Exercises

'...' 1. How much shorter can you make the code from Figure 1.1? What if you completely ignore considerations about the readability of the code?
2. Consider the following compact 2-line alternate implementation of the multi-headed self attention function:

```
def self_attn2(x, Q, K, V, P, m):
    w = softmax(np.where(m[..., None], np.einsum('nj,ml,hjk,hlk->nmh', x, x, Q, K), -1e20)
    return np.einsum('nmh,mj,hjk,hdk', w, axis=1), x, V, P)
```

What are the differences vs the implementation from Figure 1.1?

3. Can you spot the glaring inefficiency in our naive implementation of `decode` from Figure 1.2? Fix it. Hint: are some computations repeated when decoding more than one token?

♙ 4. Re-implement the Transformer (Figure 1.1) using Numpy's matrix multiplication operator `"@"` instead of `einsum`s wherever possible.

5. Re-implement the Transformer (Figure 1.1) in a different language, e.g. C. Try to make it as short and as efficient as possible.

♙ 6. Change the `decode` function to perform temperature sampling.

♙ 7. Note that the `decode` function breaks if you attempt to decode too many tokens. Change it slightly so that it uses a maximal sliding window of the last `block_size` tokens and can be called to produce an arbitrary number of tokens.

♙ 8. *Mechanistic interpretability is hard.* Write a function, `scramble_transformer` (see signature below), that gets a set of Transformer weights (in the same way as the input of the `transformer` function from Figure 1.1): `tok_emb`, `pos_emb`, `out_emb`, and `lnf`, as well as an array of layers, each consisting of `Q`, `K`, `V`, `P`, `M1`, `b1`, `M2`, `b2`, `ln1`, and `ln2` for each of the layers, and an orthonormal matrix M, and returns a new set of weights, `new_tok_emb`, `new_pos_emb`, `new_out_emb`, `new_lnf` and `new_Q`, ..., `new_ln2` for each layer, such that (1) `new_tok_emb[i]` $= M \cdot$ `tok_emb[i]` for all i, and (2) the Transformer performs *exactly the same function as before*. Which parameters need to change? What happens if M is not orthonormal?

```
def scramble_transformer(tok_emb, pos_emb, out_emb, layers, lnf, M):
    assert np.allclose(M @ M.T, np.eye(tok_emb.shape[1]))  # M is orthonormal.
    ...
    assert np.allclose(new_tok_emb, np.einsum('vd,dk->vk', tok_emb, M))
    return new_tok_emb, new_pos_emb, new_out_emb, new_layers, new_lnf
```

'...' 9. In this book we'll tackle some basic algorithms, like printing a fixed message, looking up memorized patterns, searching for patterns, sorting a list, and performing addition (you'll see the formal definitions of each of the problems at the beginning of their respective chapters). For now, try to make some educated guesses: (1) How many layers will we

be using on average? (2) How many model dimensions will we be using on average? (3) How many parameters? (4) What fraction of those will turn out to be zeroed? (5) Which modules (attention vs MLP) will we be using more often? Make a note of your guesses and when you finish the book check how many you got right.

♛♪'...' 10. Think about the Transformer architecture - what modifications would you try to implement to make it better? Answer this question once again after finishing this book.

Part II

Transformer Programs

Chapter 2

Hello World

If you can write "Hello World" you can change the world.

RAGHU VENKATESH

The first program we'll implement, Hello World, gets no input after the `<bos>` token, and simply outputs the fixed string "Hello World!". Formally:

Input `[<bos>]`.

Output: `[H, e, l, l, o, , W, o, r, l, d, !, <eos>]`.

Beware - spoilers ahead! If you'd like to implement Hello World yourself, now would be a good time to do so. Read on for the solution.

Observe that the output of this simple program is a function of the absolute position alone. We'll soon observe that as such, it can be implemented with a 0-layer Transformer. Being the very first program that we'll tackle, and to more gently introduce us to Transformer programming, we'll start by implementing Hello World on a much simpler architecture, the *Simpleformer*.

```python
def simpleformer(seq, pos_emb, out_emb):
  x = pos_emb[np.arange(len(seq))]
  return np.einsum('nd,vd->nv', x, out_emb)

def decode_simpleformer(tokens, pos_emb, out_emb, eos):
  while True:
    next_tok = np.argmax(simpleformer(tokens, pos_emb, out_emb)[-1])
    tokens = np.append(tokens, next_tok)
    if next_tok == eos: break
  return tokens
```

Figure 2.1: The Simpleformer.

2.1 A Simplified Implementation ϟ

The Simpleformer is identical to a 0-layer Transformer, with two additional simplifications: (1) there are no layer normalizations, and (2) the inputs are just the position embeddings - i.e., there are no input token embeddings. Figre 2.1 shows an implementation.

In other words, if we denote by o_j the output embedding for the jth token, the output of the Simpleformer for position p_i is simply $argmax_j(p_i \cdot o_j)$. If we simply guarantee then that $o_j \cdot o_j > o_j \cdot o_i$ for all i, j, an implementation of Hello World on the Simpleformer would simply consist of setting $p_i = o_{t_{i+1}}$, where t_j denotes the token at position j in the output sequence. See Figure 2.2. At this point you might say that these positional embeddings are not really encoding positions at all - they are encoding tokens. That's a great observation, but note that had we a trained a Transformer, with learned positional embeddings, the exact same the thing would happen there as well.

What is the minimum number of dimensions needed for this guarantee? Obviously one dimension is not enough. Indeed, with a single dimension we could only represent two symbols - the largest positive and the smallest negative. What about two dimensions? Two dimensions do work great in our degenerate Simpleformer. Indeed, we can for example assign a unique 2D representation to each of the tokens of the string "Hello World!" and the `<bos>` and `<eos>` tokens as points on the unit circle (Figure 2.3). With our embeddings being points on the unit circle, taking the dot product between two of them amounts to computing the cosine of the angle between the vectors, and thus the above guarantee of $o_j \cdot o_j > o_j \cdot o_i$ for $i \neq j$ holds. The actual order of the symbols inside the unit circle, as well as the distances between them don't really matter for now, as long as they are far enough from each other to avoid numeric precision issues, and we can assign the points arbitrarily

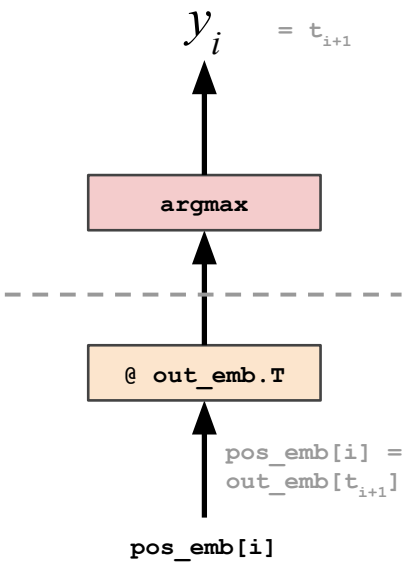


Figure 2.2: The Simpleformer architecture. The gray text denotes our trivial solution for Hello World on the Simpleformer.

(in later sections, the order and exact distances will matter, see Section 3.2).

Putting this all together, you can find a simple construction for Hello World on the Simpleformer, following this design and using 2 model dimensions, in Figure 2.4.

```
message = ["<bos>"] + list("Hello World!") + ["<eos>"]
tokenizer = {s: i for i, s in enumerate(set(message))}
detokenizer = {i: s for s, i in tokenizer.items()}
tokenize = lambda s: np.array([tokenizer[c] for c in s])
detokenize = lambda a: "".join([detokenizer[i] for i in a])

t = np.linspace(0, 6, len(tokenizer))  # 6 = 2 pi - epsilon.
out_emb = np.stack((np.cos(t), np.sin(t)), axis=1)
pos_emb = out_emb[tokenize(message[1:])]
output = detokenize(
  decode_simpleformer(tokenize(["<bos>"]), pos_emb, out_emb, eos=tokenizer["<eos>"])[1:-1])

assert output == "Hello World!"
```

Figure 2.4: An implementation of Hello World on the Simpleformer.

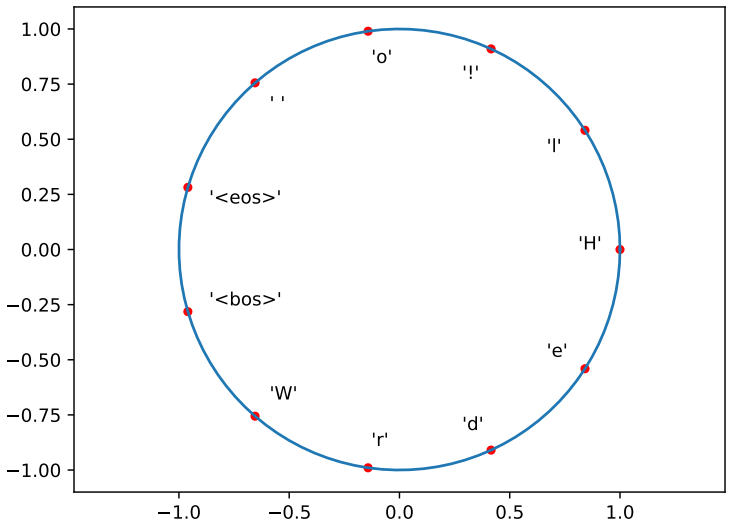


Figure 2.3: Assigning each of the tokens a unique representation on the 2D unit circle.

2.2 The Full Construction

The greater the obstacle, the more glory in overcoming it.

MOLIÈRE

We'll now turn our attention back to the full Transformer, where we will need to overcome the two "obstacles" above: layer normalizations and input token embedding. A 0-layer Transformer, which can implement functions of the current token and position only, as depicted in Figure 2.5 will work. Similarly to our construction on the Simpleformer, it will suffice to somehow encode in each absolute position the next token to decode.

Overview

As per the figure, a 0-layer Transformer adds together `tok_emb[`t_i`]` and `pos_emb[i]`, passes the result through the final `layer_norm` (as there are no layers!), and multiplies the normalized result by `out_emb` (which equals `tok_emb` transposed in our case) to get the logits. Finally, the greedy decoder applies `argmax` to the logits to get the output y_i.

As in the construction for the Simpleformer, we'd like y_i to equal the next token to decode t_{i+1},

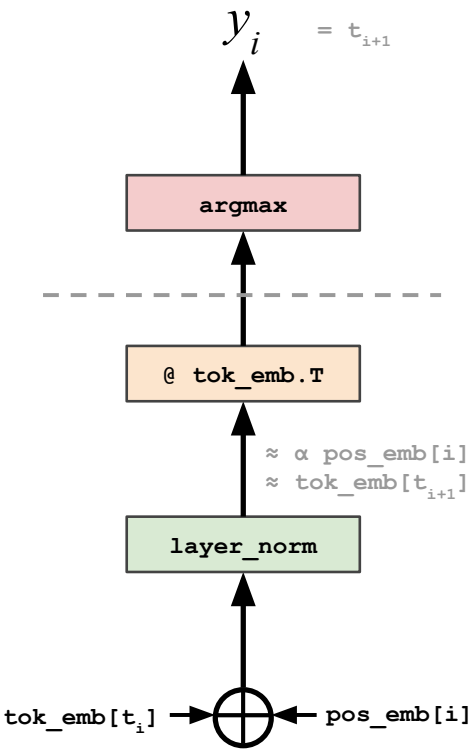


Figure 2.5: The 0-layer Transformer we'll use for the Hello World program. The part above the dashed line is the `decode` function. The gray text shows the desired values.

i.e., we'd like `argmax(logits)` to equal t_{i+1}. For that, the output of the final `layer_norm` should be closer to `tok_emb`$[t_{i+1}]$ than to the embedding of any other token.

Ignoring the Input Tokens

If we were not tying together the input and output embedding (`tok_emb` and `out_emb`), we could simply zero out the token embeddings, and then the position embeddings would directly flow into the final layer normalization. For slightly increased challenge though, and to follow [Press and Wolf, 2017], we *will* be tying the weights, so we can't zero out `tok_emb` (otherwise `out_emb` would be zeroed out too!). We'll therefore first have to deal with the input token embeddings "dirtying" our position embeddings. To nullify the effect of the addition of the token embedding, we could simply set `pos_emb[i] = tok_emb`$[t_{i+1}]$ `- tok_emb`$[t_i]$. Another option, would be to set `pos_emb[i] =` $N\cdot$ `tok_emb`$[t_{i+1}]$ for a very large N. We then have, `layer_norm(tok_emb`$[t_i]$ `+ pos_emb[i]) ≈ layer_norm(pos_emb[i])`. In our construction for HELLO WORLD we will use the latter, with $N = 1,000,000$. See Section 3.4 for a deeper dive on this topic.

Dealing With Layer Normalization

To finish the construction, we need to make sure that the above holds even after layer normalization. In fact, it's as easy to guarantee something even stronger - that `layer_norm(pos_emb[i]) = pos_emb[i]`, i.e., that the position embeddings (and likewise the token embeddings, which are taken from the same set according to our construction above) will remain fixed under layer normalization.

Unfortunately, if we used 2 dimensions like in the construction for the Simpleformer above, the final `layer_norm` (the green box in Figure 2.5) would collapse all of our 2D encoding (and actually all of $\mathbb{R}^2$!) into just two points*, so our encoding won't satisfy the desired quality above (or most desired qualities for that matter - it will be a pretty boring encoding!). See Figure 2.6. What is the minimum number of dimensions that are needed with a standard Transformer with layer normalization? Luckily, as we'll now observe, 3 dimensions are enough with a simple construction in $\mathbb{R}^3$.

2.3 3D Positional Encoding

As noted above, layer normalization reduces the dimension of a space by 2, meaning that it would collapse the entirety of the 2D unit circle (or actually the entirety of $\mathbb{R}^2$, Figure 2.6). Luckily, 3 dimensional positional encoding *are* enough to work. Which subspace of

* We'd also get some undefined behavior on the line $y = x$.

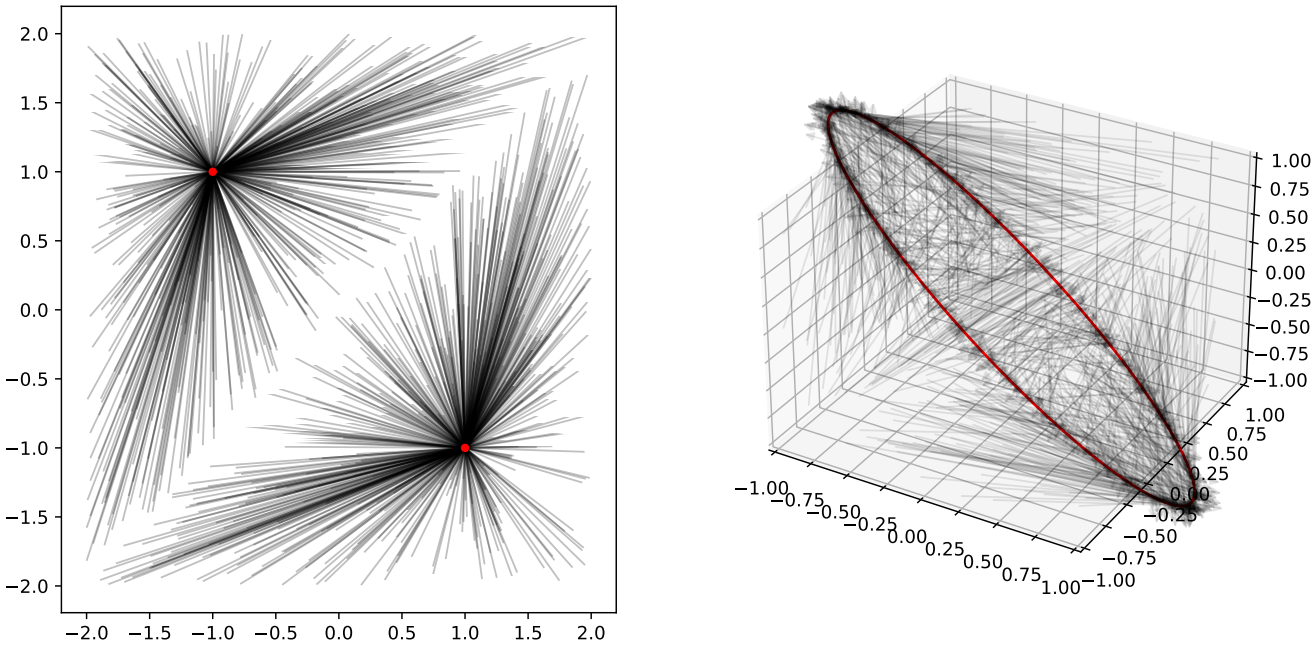


Figure 2.6: Layer normalization collapses all of $\mathbb{R}^2$ into the points (1, -1) and (-1, 1), allowing us to represent a maximum of 2 points in $\mathbb{R}^2$ and all of $\mathbb{R}^3$ into the $\sqrt{3} - circle$ in the plane perpendicular to (1, 1, 1), enabling a rich representation there.

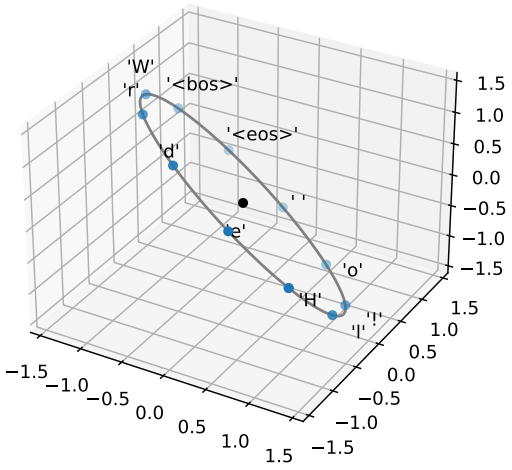


Figure 2.7: Assigning each of the tokens a unique representation on the 2D $\sqrt{3}$-circle in $\mathbb{R}^3$ perpendicular to $(1, 1, 1)$. See Section 2.3.

$\mathbb{R}^3$ should we use? To satisfy the first condition of layer normalization (of zero mean, i.e., $\sum_i v_i = 0$) we'll restrict ourselves to the 2D-plane in $\mathbb{R}^3$ that's perpendicular to the vector $(1, 1, 1)$. To satisfy the second condition (of unit variance, i.e., $\sum_i v_i^2 = 3$) we'll restrict ourselves to points of magnitude $\sqrt{3}$ in that plane. To summarize, all points on the circle with radius $\sqrt{3}$ on that plane perpendicular to $(1, 1, 1)$ are fixed under `layer_norm`. Finally, to minimize numerical instability we'd like the points to be as far from each other as possible, so we'll make sure that our encodings are evenly spaced on the above circle (as we'll later see in Section 3.2 this will be important for another reason as well - linear shiftability). Figure 2.7 shows our construction of 3D positional encodings for the tokens in the string "Hello World!" (and the `<bos>` and `<eos>` tokens as well).

At this point you might be asking why I chose to call this construction 3D *positional* encodings - there is nothing very positional about them! Indeed, we are using them here mostly for representing tokens, not positions. This name will hopefully make more sense in Section 3.2, where we'll observe that making sure that the points are arranged in order and with fixed distances on the unit circle is a very helpful construction for positional encoding. For now, just don't dwell too much on the name.

For a simple code implementation of these 3D positional encoding, we'll first use a para-

meterization $p(t) : [0, 2\pi] \to \mathbb{S}^1$, that sends a point $t \in [0, 2\pi]$ to $(cos(t), sin(t))$. We then embed $\mathbb{S}^1 \subset \mathbb{R}^2$ in $\mathbb{R}^3$, and finally we send this unit circle to the rotated and scaled one above using a linear transformation, so it's enough to define it on the vectors of any basis of $\mathbb{R}^3$. Specifically, we'll construct a rotation that sends $(0, 0, 1)$ to $(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}})$, $(1, 0, 0)$ to an arbitrary point on the rotated unit circle $(\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}, 0)$, and finally sends $(0, 1, 0)$ to an orthogonal point on the rotated unit circle $(\frac{1}{\sqrt{6}}, \frac{1}{\sqrt{6}}, \frac{-2}{\sqrt{6}})$. Scaling the resulting point by a factor of $\sqrt{3}$ gets us the desired embedding. See Figure 2.8.

```
def pos_enc_3d(t):
    x, y = np.cos(t) / (2 ** .5), np.sin(t) / (6 ** .5)
    return np.array([x + y, -x + y, -2 * y]) * (3 ** .5)

def pos_enc_3d_array(n):
    return np.stack([pos_enc_3d(2 * np.pi * t / n) for t in range(n)])
```

Figure 2.8: Our construction of 3D (linearly shiftable, see Section 3.2) positional encodings, that are fixed under `layer_norm`.

2.4 Putting It All Together

Putting it all together, the final part of the construction of Hello World consists of setting the position encoding such that each position will decode to the corresponding token. Similarly to our construction of Hello World for the degenerate Simpleformer above, we'll set `pos_emb[i]=tok_emb[`t_{i+1}`]`$\cdot N$ where t_i is the ith token in the string to decode ("Hello World!") and N is a large number (we'll use 1,000,000). Denoting $P_i =$ `pos_emb[i]` and $T_t =$ `tok_emb[t]`, we get:

$$x_i = T_{t_i} + P_i = T_{t_i} + N \cdot T_{t_{i+1}} \approx N \cdot T_{t_{i+1}}$$

And that's it - we have a standard Transformer that prints "Hello World!"! Figure 2.9 shows the full Hello World Transformer in code.

```python
block_size = len(message) - 1
vocab_size = len(tokenizer)
params = base_params(vocab_size, block_size, d_model=3)
params['tok_emb'] = params['out_emb'] = pos_enc_3d_array(vocab_size)
params['pos_emb'] = params['out_emb'][tokenize(message)[1:]] * 1_000_000

output = detokenize(decode(tokenize(['<bos>']), params, eos=tokenizer['<eos>'])[1:-1])

assert output == 'Hello World!'
```

Figure 2.9: The Hello World Transformer.

Excluding the final `layer_norm`'s `beta` and `gamma`, and the output embedding (which are here tied to the input embedding), our Transformer has 13 positions and 11 unique tokens, leading to a total parameter count of $(13 + 11) \cdot 3 = 72$. Note though, that this number assumes a message-dependent tokenizer. If instead we used a generic ASCII tokenizer (and e.g. '\0' as both `<bos>` and `<eos>` tokens), we'd have $(13 + 256) \cdot 3 = 807$ total parameters. With this generic ASCII tokenizer, we could use the construction above to output any fixed message, such that for messages where $|message| \gg 256$ we'd need $\sim 3 \cdot |message|$ parameters. See Exercise 2 for implementations that are more efficient in the sparse sense.

2.5 Exercises

♙ 1. Write a simple program, on a standard 0-layer Transformer, using at most $3 \cdot$ `vocab_size` parameters, that indefinitely repeats the last token from the input.

♙ 2. *Optimizing for sparse parameters.* (1) Repeat Exercise 1 but use only `vocab_size` *non-zero* parameters. (2) Construct the full Hello World program using only `vocab_size` + `block_size` *non-zero* parameters. (3) How many total parameters, zero and non-zero, did you need for these constructions?

♪ 3. Actually train a Transformer, i.e. with a standard optimization procedure, to output "Hello World!". How small can you make it?

♙♪ 4. Modify the construction of Hello World from this chapter to instead output "Hel Word!" (i.e. `[H, e, l, , W, o, r, d, !]`), but use a **zeroed out** `pos_emb` array. You can untie `tok_emb` and `out_emb`.

♛ 5. Repeat Exercise 4, i.e. use a zeroed out `pos_emb` array, but produce the original "Hello World!" output (i.e. `[H, e, l, l, o, , W, o, r, l, d, !]`).

♙ 6. Show that if our Transformer implementation used RMSNorm [Zhang and Sennrich, 2019], instead of standard LayerNorm, it could be made more efficient - specifically we'd get

back one free dimension. How many parameters would our construction need in that case?

7. Implement Hello World using only *integer* parameters.
8. Implement Hello World using only *positive* parameters.

Chapter 3

Lookup Table

Low-level programming is good for the programmer's soul.

JOHN CARMACK

Never memorize something that you can look up.

ALBERT EINSTEIN

We'll now implement a simple Lookup Table - i.e. we'll make our Transformer memorize a mapping between keys (fixed-length sequences of tokens) and values (single tokens). To make things slightly more interesting, we'll allow prefixing the input key sequence to look up with an arbitrary string. This prepended prefix will be ignored, and the look up will only consider the fixed-length suffix of the input. More precisely, given n fixed key-value pairs, $(k_1, v_1), \ldots, (k_n, v_n)$, such that the keys k_i are strings of length l and the values are tokens $k_i \in \Sigma^l$, $v_i \in \Sigma$ (Σ denotes our vocabulary), our goal for input string $x_1, \ldots, x_t$ s.t. $x_{t+1-l}, \ldots, x_t = k_j$ would be to output v_j. I.e., if the last l tokens of the input match key k_j, we will output v_j. We'll leave as undefined behavior cases where the last l tokens don't match any sequence, or match more than one sequence.

For example, given the lookup table represented by the pairs `([1, 2, 3], 4)` and `([4, 5, 6], 7)`:

Key	Value
`1, 2, 3`	`4`
`4, 5, 6`	`7`

Lookup Table will behave as follows:

Input: `[1, 2, 3, 4, 5, 6]`.

Output: `7`.

The main construction in this chapter we will focus on an easier version of Lookup Table where the keys and values are chosen at random. We'll show that our simple construction will work perfectly with high probability for random lookup tables, but it will not work for *all* possible tables. Exercise 11 asks you to construct an adversarial example to our simple construction, and in Exercise 12 you will develop a construction that works for all cases.

Beware - spoilers ahead! If you'd like to implement Lookup Table yourself, now would be a good time to do so. Read on for the solution.

Here's a quick sketch of our construction, which we will make precise later: we will have the last token read the preceding $l - 1$ tokens, generate and store a hash of them, such that the hash will be unique for each entry in the table, and arrange the output embedding to produce the desired value for each of the hashes.

We note that Lookup Table is unique amongst the programs in this work in that it is very parametric. While the other programs also depend on some parameters, like the maximum block size or the size of the vocabulary, these usually don't alter our construction in a substantial way, whereas Lookup Table completely changes given an external input - namely the table to memorize. To that end, we'll actually be solving a small optimization problem in Section 3.6. Following this construction is completely optional, but note that in Sections 3.1-3.4 we'll describe four basic blocks that we will continue using throughout the rest of this work. With that in mind, feel free to skip the end of this chapter, but make sure you understand the ideas in Sections 3.1-3.4.

3.1 Attention Hardening

An extremely useful tool for us will be reading memory from previous tokens *without noise*. Unfortunately, our Transformer employs standard soft attention:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

To read memory without noise, we'd like to simulate hard attention, i.e. replace the softmax with regular max (and then we can drop the $\frac{1}{\sqrt{d_k}}$ factor as well):

$$HardAttention(Q, K, V) = max(QK^T)V$$

Due to the exponential in the softmax, hard attention is easily approximated by multiplying Q (or equivalently K) by a large number N:

$$HardAttention(Q, K, V) \approx Attention(NQ, K, V) = Attention(Q, NK, V)$$

We'll often use $N = 1,000,000$.

3.2 Attending to Relative Positions

A Transformer decoder can only attend to the left. That means that whenever we need to process several tokens, we'll have to carry out the computation in the position of the right most token being considered. Most of our programs, including Lookup Table will be sending information from all tokens involved in the computation right-ward, by having tokens on the right attend to one or more tokens to their left and copying some information.

Previously, when constructing Hello World, we used the position embedding to encode tokens directly. Now, more generally, we'd like to use the position embedding for their intended purpose, and e.g. be able to read information from a token that is some specific number of positions to our left. We'll now construct the mechanism for performing such an attention operation.

Definition 3.1 (Linearly Shiftable Encoding)**.** *An encoding $E : \mathbb{Z}_n \to \mathbb{R}^d$, over a vocabulary of size n, is **linearly shiftable** if for every $\delta \in \mathbb{Z}$ there exists a linear transformation $T_\delta : \mathbb{R}^d \to \mathbb{R}^d$ such that for all integer $j \in \mathbb{Z}_n$ where $0 \leq j+\delta < n$ we have $T_\delta(E(j)) = E(j+\delta)$. We'll call T_δ the **shifting transformation** of E by δ.*

I use the notation $\mathbb{Z}_n$ to denote the ring of integers mod n, i.e. $\mathbb{Z}/n\mathbb{Z}$. In other words $\mathbb{Z}_n$ is a shorthand for the set $\{0, 1, \ldots, n-1\}$.

```python
def pos_3d_enc_rotation_matrix(theta):
    T = np.empty((3, 3))
    T[0] = pos_enc_3d(0)
    T[1] = pos_enc_3d(1)
    T[2] = pos_enc_3d(2)

    S = np.empty((3, 3))
    S[0] = pos_enc_3d(0 + theta)
    S[1] = pos_enc_3d(1 + theta)
    S[2] = pos_enc_3d(2 + theta)

    return np.linalg.solve(T, S)
```

Figure 3.1: The shifting transformation T_δ where $\theta = \frac{2\pi\delta}{n}$.

As pointed out by [Vaswani et al., 2017] the original Transformer's sinusoidal embeddings are linearly shiftable. See optional Chapter 7 for further thoughts on the original Transformer's embeddings.

We are actually interested in a slightly weaker version of linear shiftability:

Definition 3.2 (Weak Linearly Shiftable Encoding)**.** *An encoding $E : \mathbb{Z}_n \to \mathbb{R}^d$ is **weak linearly shiftable** if there exists a linear transformation M that projects E into a linearly shiftable embedding.*

Note that it is possible for an embedding to not be linearly shiftable, but be weak linearly shiftable (e.g. any linearly shiftable encoding embedded in a higher dimensional space, with the new dimensions set to some non linear function, like a counter).

For both linearly shiftable encoding (Definition 3.1) and weakly linearly shiftable encoding (Definition 3.2) we'll only be interested in non-degenerate encodings, i.e. where $|E(\mathbb{Z}_n) = n|$.

Given a weak linearly shiftable positional encoding, attending to a relative position is easy - we simply set the key projection $K = M$ (from Definition 3.2) and the query projection $Q = T_\delta M$, so we get $(Kv, Qu) = (E(p_v), E(p_u + \delta))$, which, assuming the encoding is normalized, attains its maximum value exactly when $p_v = p_u + \delta$. To keep things simple, we'll often set M to be the natural projection into an axis-aligned subspace (i.e. we'll simply zero out some of the dimensions). With the attention hardening trick above and assuming the encodings are normalized, in order to read from a relative position without noise, it is then enough to ensure that the positional embeddings are pairwise independent.

While evenly spaced points on the 2D unit circle are linearly shiftable and with attention

hardening would allow encoding very large blocks without floating point precision issues, as discussed in Section 2.3, 2D encoding are not fixed under layer normalization. We note though that if we make sure the 3D positional encoding we constructed in Section 2.3 are evenly spaced, they are both linearly shiftable and fixed under layer normalization. That's the construction we will use. Specifically, given δ we set $\theta = \frac{2\pi\delta}{n}$, and construct the shifting transformation T_δ as in Figure 3.1.

3.3 Bypassing Layer Normalization

Recall that our Transformer implementation uses PreLN* (see Figure 3.2). As will soon become evident, and extending the ideas from our construction for HELLO WORLD, we'd like to construct our hidden representations such that layer normalization would leave them unchanged. We'll now develop several tools to that effect.

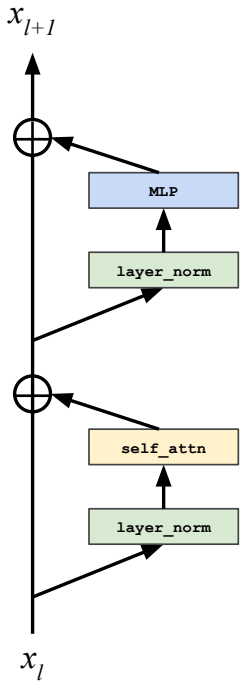


Figure 3.2: An illustration of `Transformer_layer` using PreLN.

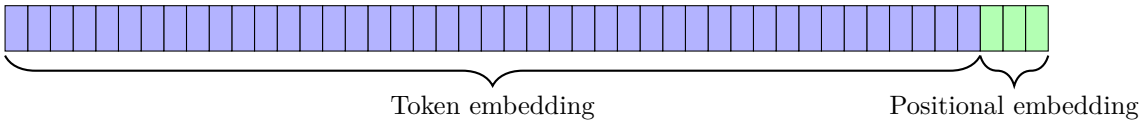
We observe that for a vector $v \in \mathbb{R}^d$, if $\sum_i v_i = 0$ and $\sum_i v_i^2 = d$ then `layer_norm(`v`)` $= v$ as desired. Immediately follows that for vectors $v \in \mathbb{R}^{d_1}, u \in \mathbb{R}^{d_2}$, if `layer_norm(`$v$`)` $= v$

* Exercise 7 asks you to implement LOOKUP TABLE with PostLN.

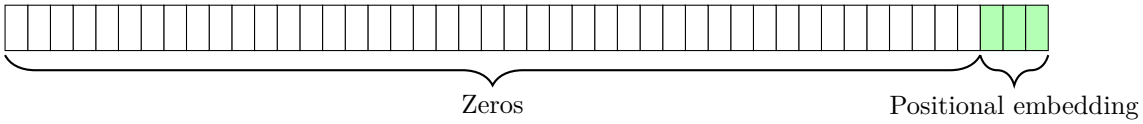
and `layer_norm`$(u) = u$, then `layer_norm(concatenate(`v`, `u`))` = `concatenate(`v`, `u`)`, i.e. if we construct our representation as a concatenation of blocks that are left unchanged by `layer_norm`, then the representation itself will be left unchanged by `layer_norm`.

Theorem 3.1. *If* $v \in \mathbb{R}^{d_1}, u \in \mathbb{R}^{d_2}$ *are fixed under layer normalization in* $\mathbb{R}^{d_1}$ *and* $\mathbb{R}^{d_2}$ *respectively, then* `concatenate(`v`, `u`)` *is fixed under layer normalization in* $\mathbb{R}^{d_1+d_2}$.

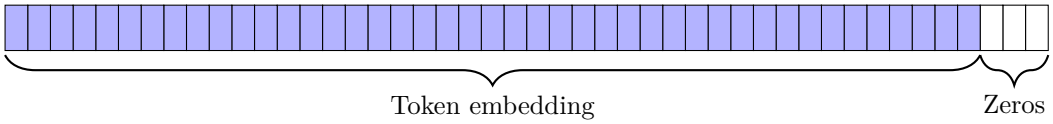
We will then construct our hidden representation for LOOKUP TABLE following this schematic:



To that end, we'll make sure that `pos_emb` is fixed under layer normlization and properly padded:



And similarly for `tok_emb`:



Note that the above diagrams are for illustration purposes only, and specifically, the number of dimensions that we'll actually use in LOOKUP TABLE for the token embedding will depend on the parameters of the table to memorize, as we'll soon observe. Later, in Section 4.1, we'll construct a slightly more sophisticated construction that will allow for additional storage space.

3.4 Cleaning Up

Always leave the campground cleaner than you found it.

A BOY SCOUTS RULE

In most of our programs, including Lookup Table, we'd like to perform some calculation and retain the result of that calculation for downstream processing. For example, for Lookup Table we'd like to generate the signature of the preceding tokens to be fed to the unembedding matrix. Unfortunately[†], due to the residual connections, the result of the calculations performed by our attention or MLP sub-layers will be added to the input of the layer, instead of overriding it. We'll explore two simple tricks to remedy the situation (we've already mentioned those more briefly in Chapter 2).

The first method, *proper cleanup*, consists of using hardened attention and attend to ourselves ($K = \mathbb{I}, Q = N\mathbb{I}$, see Section 3.1), and set the V and P matrices to copy and negate our contents: $V = \mathbb{I}, P = -\mathbb{I}$. Note that if the head size d_{head} is smaller than d_{model}, we might need several heads to perform this cleanup. Also, note that if we'd only like to free up part of the space of our hidden representation, we could obviously do that as well by only copying the parts to be cleaned up. Due to downstream `layer_norm`, instead of actually erasing the content to be deleted, we'd usually like to set it to layer normalization resistant padding, as we will see in Section 4.1 (or, more commonly, override it with the result of our calculation).

One caveat of proper cleanup, is that, unless we are attending to ourselves anyway, we'd need to "waste" an attention head on this clean up (potentially more than one, depending on d_{model}, d_{head}, and the size of the block to be cleaned). Fortunately, in most of our programs we'll be attending to ourselves anyway, so we could just add $\mathbb{I}$ and $-\mathbb{I}$ to V and P for the appropriate head, and get proper cleanup "for free". Also observe that even in case where we lost position information (or didn't start with them to begin with because they are otherwise unnecessary) we can always attend to ourselves perfectly as above.

The second method, *messy cleanup*, is even simpler, and exploits downstream layer normalization. It consists simply of multiplying the result of the calculation we'd like to overwrite our representation with by a large constant. Specifically, denote the result of the calculation by u and the original data by v. Assuming $\sum_i u_i^2 \gg \sum_i v_i^2$, then $\texttt{layer\_norm}(u + v) \approx \texttt{layer\_norm}(u)$. This is the approach we took when constructing Hello World in Section 2.

[†] Well, it is only unfortunate *for us*. Generally the residual connection is a critical mechanism to help the optimization procedure propagate gradients.

3.5 Simplified Construction for an Untied Transformer

Before we describe the construction for a Transformer where the input and output embeddings are tied (Section 3.6), let's start with a simpler construction for a Transformer where the output and input embeddings aren't tied. For this simpler construction we will not care much about efficiency (i.e. number of parameters).

The original Transformer implementation [Vaswani et al., 2017], follows [Press and Wolf, 2017] and tie the input embedding and the pre-softmax output embedding. The motivation being, quoting [Press and Wolf, 2017]:

> "In both matrices, we expect rows that correspond to similar words to be similar: for the input embedding, we would like the network to react similarly to synonyms, while in the output embedding, we would like the scores of words that are interchangeable to be similar."

We note that for the randomized sequences of LOOKUP TABLE (as well as those of most of the rest of the programs in this work), this motivation does not hold. For this easier construction then, we will untie the weight matrices (i.e. we will not force `out_emb` to equal `tok_emb`).

A first idea would be to initialize `tok_emb` randomly, and then, by attending to the left (Section 3.2), copy the embeddings of the preceding tokens into a concatenated representation, and setup the output embedding to map from the concatenated embeddings to the desired output. For the copying to be possible, we'd need to allocate enough redundant storage in the destination (right) token to contain the needed information content from the left tokens. For example, if we need d dimensions of information per token from k tokens, we could allocate (i.e. reserve in the hidden representation) $k \cdot d$ dimensions and copy the relevant data from k tokens, concatenated one after the other. A second and less wasteful approach could be to *hash* the information from the preceding tokens together into a more compact representation. A simple hash of k vectors $v_1, \ldots, v_k \in \mathbb{R}^d$ into a d dimensional representation, might consist of using a set of matrices $G_1, \ldots, G_k \in \mathbb{R}^{d \times d}$, one per relative position, and define `hash(` $v_1, \ldots, v_k$ `)` `=` $\sum_i G_i v_i$. For example, G_i could be random matrices $G_i \sim \mathbb{N}(0, \mathbb{I})$ or random orthogonal matrices.

With this idea in mind we'll make use of a single layer Transformer for our simplified construction. As a matter of fact, we'll only be using the attention module from that layer, and won't be needing the MLP part at all (we'll make sure it provides no contribution to the residual). We will initialize the position embeddings to the standard ones we discussed above and the token embeddings to normalized random points, both with proper padding as

```python
def build_untied_transformer(ks, vs, vocab_size, block_size, prefix_len, tok_dims):
    d_head = tok_dims + 3
    n_heads = prefix_len
    d_model = d_head  # We don't need the standard d_head * n_heads.

    params = base_params(vocab_size, block_size, d_model)

    params['pos_emb'] = np.pad(pos_enc_3d_array(block_size), ((0, 0), (tok_dims, 0)))
    params['tok_emb'] = np.pad(
        layer_norm(np.random.normal(0, 1, (vocab_size, tok_dims))), ((0, 0), (0, 3)))
    hash_matrices = np.random.normal(0, 1e-1, (prefix_len, tok_dims, tok_dims))

    layer0 = base_layer(n_heads, d_model, 0, d_head)
    params['layers'].append(layer0)
    for head in range(n_heads):
        theta = -head * 2 * np.pi / block_size
        layer0['Q'][head, -3:, -3:] = pos_3d_enc_rotation_matrix(theta) * 1e8
        layer0['K'][head, -3:, -3:] = np.eye(3)
        layer0['V'][head] = np.eye(d_model)
        layer0['P'][head, :tok_dims, :tok_dims] = hash_matrices[head].T
    layer0['P'][0] -= np.eye(d_model)  # Clean up residual.

    causal_mask = np.tril(np.ones((prefix_len, prefix_len)))
    for k, v in zip(ks, vs):
        x = params['tok_emb'][k] + params['pos_emb'][np.arange(len(k))]
        params['out_emb'][v] += transformer_layer(x, **layer0, mask=causal_mask)[-1]

    return params
```

Figure 3.3: The construction for LOOKUP TABLE in the simplified case of a Transformer with untied input and output embeddings.

above. We will then construct the relative attending and hashing mechanism as described above, and finally, we'll accumulate in the output embedding for each symbol the sums of the hashes of the keys leading to it. See Figure 3.3 for the construction in this simplified case.

3.5.1 Analysis ↯

There should be no such thing as boring mathematics.

EDSGER W. DIJKSTRA

So, why does this even work at all?

Consider what happens if we initialize `tok_emb` with random points on the $\sqrt{d}$-sphere in R^d, and pick G_i to be random orthogonal matrices. Then[‡], set `out_emb[i]` $= o_i$ to be $\sum_{j|v_j=i} hash(k_j)$, where $hash(k_j) = \sum_{p=1,\ldots,l} G_p \cdot \texttt{tok\_emb}(k_{j,p})$.

Take a specific entry in our lookup table, k, v. Say we have c entries in our lookup table with value v ("collisions"). What's the probability of correctly outputting v for input k? We'd want $(\texttt{layer\_norm}(hash(k)), o_i) > (\texttt{layer\_norm}(hash(k)), o_j)$ for all $j \neq i$. Now, notice that in high enough dimensions, $\texttt{layer\_norm}(hash(v)) \approx \alpha \cdot hash(v)$ for some α that only depends on d not v (see Exercises 3 and 4), so we'd want $(hash(k), o_i) > (hash(k), o_j)$ for all $j \neq i$.

Now, making the great simplifying assumption that $k_{i,j}$ and $k_{s,t}$ are independent, as is the case for *random* lookup tables (specifically where the keys are randomly chosen and the vocabulary size is large), for $j \neq i$ we have $(hash(k), o_j)$ distributed approximately like the sum of $c \cdot l^2$ dot-products between two random vectors on the d dimensional sphere ($hash(k)$ is approximately the sum of l independent vectors on the sphere, and o_j is approximately the sum of $c \cdot l$ vectors on the sphere). The dot product between two random vectors on the $d-1$-dimensional sphere in $\mathbb{R}^d$ is in turn approximately distributed $2 \cdot Beta(\frac{d-1}{2}, \frac{d-1}{2}) - 1$ with 0 mean and approximate variance of $\frac{1}{d}$ (Exercise 5). Assuming n is large and using the central limit theorem, we finally get that the above sum is approximately distributed like $\mathbb{N}(0, \frac{c \cdot l^2}{d})$. Approximating $c \approx \frac{n}{|V|}$ we get $\sigma^2 = \frac{n \cdot l^2}{|V| \cdot d}$.

Putting it all together, choosing $d \gg \frac{n \cdot l^2}{|V|}$, we should expect to get correct decoding. Empirically, for 1,000 entries in the lookup table, each of length 5 with $|V| = 1{,}000$, we get $\frac{n \cdot l^2}{|V|} = 25$ and with 100 dimensions, we get $\approx 100\%$ correct.

In Exercise 6 you will improve the above by applying some more layer normalizations to the hashes.

‡ Recall that k_j and v_j are the keys and values to memorize from our lookup table.

3.6 Full Construction ϟ

We'll now describe the full construction for LOOKUP TABLE. As in the simplified case, here too we'll use a single-layer attention-only Transformer (i.e. we will turn the MLP off by setting all of its weights to 0, as in `base_layer` in Figure 1.3, leaving only the residual connection). Each token will consist of a unique identifier and the single attention-layer will read and incorporate into the hidden representation for each token x_i a hash of the preceding l tokens of the input $x_{i+1-l}, \ldots, x_i$. Specifically, the last input token x_t will contain a unique signature of the key to look up. We'll construct this key in such a way that if the preceding l elements equal k_j for some j, its dot-product with the token embedding table at location v_j, `tok_emb[`v_j`]`, will be maximal, ensuring that our output token will be v_j, as required. To construct the above signature, we'll need to first read the memory of the preceding l tokens, which we'll do by attending to relative positions (Section 3.2). To read l tokens, we will use l attention heads and a single layer. We could also use e.g. a single attention head and l layers, or a more sophisticated combination, but our construction turns out most efficient.

How should we construct our token embeddings and hash matrices so that they are efficient (i.e. use few parameters)? Observe that the answer depends on the entries in the (fixed) lookup table. For example, consider the degenerate case where v_i depends only on $k_{i,j}$ for a fixed j for all is, then clearly we could set $G_i = \begin{cases} 0, & i \neq j, \\ \mathbb{I}, & \text{otherwise} \end{cases}$ [§]. In order to find the optimal setup for our fixed lookup table then, we'll run a small optimization process. Note that we are not optimizing for the Transformer end to end, but just for the token embeddings and hash matrices. Importantly, note that even for this simple problem, optimizing the Transformer directly on a synthetic dataset is far from trivial - for example, we'd need to append prefixes of varying lengths to each of our examples and fill them with random (or even adverserial values - e.g. keys from other sequences), to ensure that our Transformer isn't exploiting some undesired phenomenon in the data (see Exercise 1). By optimizing just the token embedding and hash matrices, we guarantee that the Transformer will perform as desired. Let $M \in \mathbb{R}^{p \times d \times o}$ be the hash matrices, $E \in \mathbb{R}^{n \times p \times d}$ be the token embedding, k and v denote the lookup table to represent, and y a onehot encoding of v. Then the loss and its gradient can be calculated as per Figure 3.4.

We then optimize E and M, ensuring to layer-normalize E after every iteration, as in Figure 3.5. Finally, Figure 3.6 puts everything together.

§ Actually, in this case we could just use a single attention head!

```python
def logsumexp(x, axis=-1):
  mx = x.max(axis=axis, keepdims=True)
  return np.expand_dims(np.log(np.sum(np.exp(x - mx), axis=axis)), axis=axis) + mx

def loss_and_grad(E, M, k, v, y):
  V, d = E.shape
  x = E[k]  # n, l, d
  h = np.einsum('nld,ldo->no', x, M)  # n, d
  mean = h.sum(axis=1)[:, None] / (d + 3)
  h0 = h - mean
  var = ((h0 ** 2).sum(axis=1)[:, None] + 3 * mean ** 2) / (d + 3)  # n, 1
  std = var ** 0.5 + 1e-10  # n, 1
  h01 = h0 / std  # n, d
  z = np.einsum('vd,nd->nv', E, h01)  # n, V
  s = softmax(z)  # n, V
  loss =  (logsumexp(z) - z)[np.arange(len(z)), v].sum()

  dz = s - y  # n, V
  dh01 = np.einsum('nv,vd->nd', dz, E)  # n, d
  dstd = np.einsum('nd,nd->n', dh01, h0)[:, None] / (-std ** 2)  # n, 1
  dvar = dstd / (2 * (std - 1e-10))  # n, 1
  dh0 = dh01 / std + 2 * dvar * h0 / (d + 3)  # n, d
  dmean = dvar * 6 / (d + 3) * mean - dh0.sum(axis=1)[:, None]
  dh = dh0 + dmean / (d + 3)
  dM = np.einsum('no,nld->ldo', dh, x)
  dx = np.einsum('no,ldo->nld', dh, M)
  dE = np.einsum('nd,nv->vd', h01, dz)
  np.add.at(dE, k, dx)

  return loss, (dM, dE)
```

Figure 3.4: The loss and gradient calculation for Lookup Table.

```python
from tqdm import tqdm

onehot = lambda t, v: np.eye(v)[t]

def optimize(E, M, ks, vs, iters, lr=1e-3):
  vocab_size = E.shape[0]
  y = onehot(vs, vocab_size)
  with tqdm(range(iters)) as pb:
    for _ in pb:
      E[:] = layer_norm(E)
      loss, (dM, dE) = loss_and_grad(E, M, ks, vs, y)
      E -= lr * dE
      M -= lr * dM
      pb.set_description(f'{loss=}')
```

Figure 3.5: Optimizing the token embedding and hash matrices for LOOKUP TABLE.

```python
def build_lookup_transformer(ks, vs, vocab_size, block_size, tok_dims, optimization_iters):
  n, prefix_len = ks.shape
  d_head = tok_dims + 3
  n_heads = prefix_len
  d_model = d_head  # We don't need the standard d_head * n_heads.

  params = base_params(vocab_size, block_size, d_model)

  params['pos_emb'] = np.pad(pos_enc_3d_array(block_size), ((0, 0), (tok_dims, 0)))

  tok_emb = np.random.normal(0, 1e-1, (vocab_size, tok_dims))
  hash_matrices = np.random.normal(0, 1e-1, (prefix_len, tok_dims, tok_dims))
  optimize(tok_emb, hash_matrices, ks, vs, optimization_iters)
  params['tok_emb'] = params['out_emb'] = np.pad(layer_norm(tok_emb), ((0, 0), (0, 3)))

  layer0 = base_layer(n_heads, d_model, 0, d_head)
  params['layers'].append(layer0)
  for head in range(n_heads):
    theta = -head * 2 * math.pi / block_size
    layer0['Q'][head, -3:, -3:] = pos_3d_enc_rotation_matrix(theta) * 1e8
    layer0['K'][head, -3:, -3:] = np.eye(3)
    layer0['V'][head] = np.eye(d_model)
    layer0['P'][head, :tok_dims, :tok_dims] = hash_matrices[-1 - head].T
  layer0['P'][0] -= np.eye(d_model)  # Clean up residual.

  return params
```

Figure 3.6: Building the LOOKUP TABLE Transformer.

3.7 Evaluation

To test our construction, we generate a set of random lookup tables, build the Transformers as above, and measure the decoding precision, see Figure 3.8. We see that we can represent lookup tables with 10 entries with just 8 model dimensions for all tested vocabulary sizes with 100% precision, 13 model dimensions for 100 entries, and 33 dimensions for 1,000 entries (except when the vocabulary size is extremely small and our procedure fails[¶]). In terms of parameter counts, this translates to 1,408 params (283 non-zero params), 3,588 params (733 non-zero params), and 22,308 params (5,033 non-zero params) for perfectly memorizing a lookup table with 10, 100, and 1,000 entries with vocab sizes between 10 and 1,000. Not bad!

3.7.1 Some Silliness

> *If people never did silly things, nothing intelligent would ever get done.*
>
> LUDWIG WITTGENSTEIN

As noted above, representing our randomized lookup tables is a harder problem than representing more regular lookup tables, like those emerging from English n-gram models. Just for fun, let's further test our construction by taking 1,000 non-overlapping 6-grams from the text of Romeo and Juliet (which results in a vocabulary of size 1,540). As you can see in Figure 3.7 the histogram is very sharp, as is typical for natural language. If we train our Transformer on this text, we are able to perfectly decode these 1,000 6-grams with a model of just 23K parameters.

Now for the silly part - what happens if we regard this lookup table as a "soft" n-gram table, and decode new text from it? See Figure 3.9 for some results.

3.8 Exercises

♛ 1. Implement LOOKUP TABLE by using a standard optimization procedure for the *entire* Transformer - i.e. build an appropriate dataset and train a transformer to perform the LOOKUP TABLE operation on it without any manual setting of weights. *How would you guarantee that there are no adversarial prefixes that would make your model fail?*

2. Reimplement LOOKUP TABLE for the untied case (Figure 3.3) using simple concatenation instead of hash matrices.

[¶] I'm sure there is a simple explanation for not being able to memorize 1,000 with a vocab size of 10 with more than ∼20% precision - but alas it is evading me right now. Please reach out if you know why!

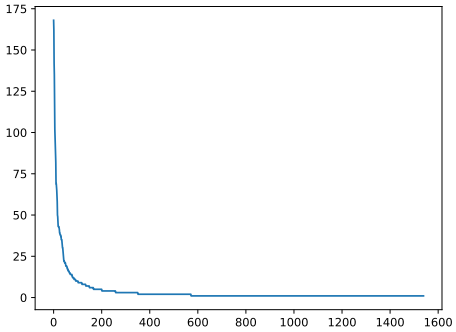


Figure 3.7: Words from "Romeo and Juliet" follow a sharp distribution.

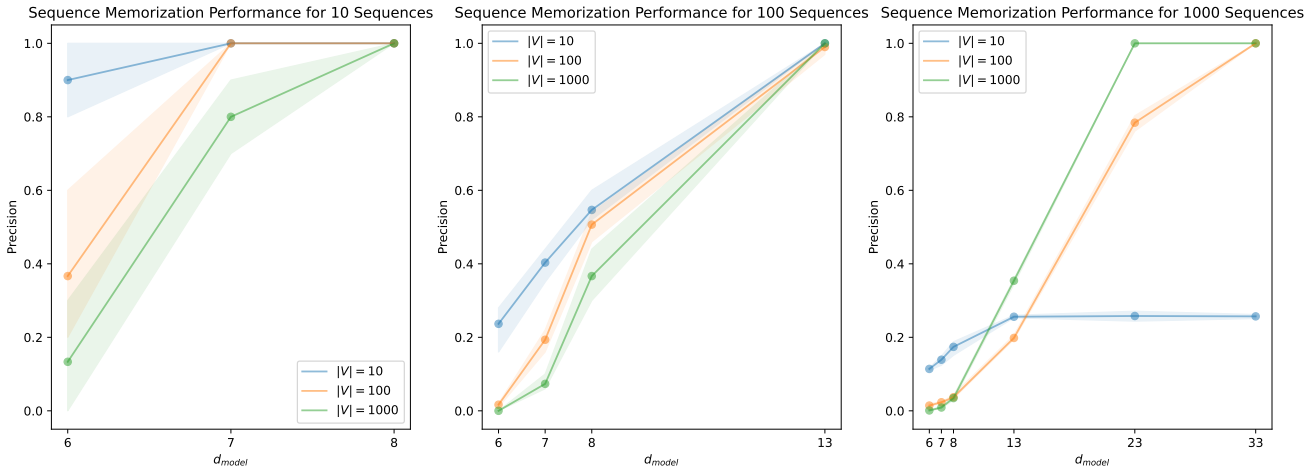


Figure 3.8: Decoding precision for various model sizes for lookup tables with 10, 100, and 1,000 entries. The shaded regions represent the minimum and maximum values for 10 runs.

```
am done mercutio tut dun s the romeo of too to meeting nurse a to of
nay that s not so mercutio i thou weak thee thou is no well i is
of long spinners legs the cover of that she in that a combin that my that
even now the frozen bosom of the thou impute the ho i is her how is
and they unwash d too tis a juliet to to him romeo she sirrah a i
say truth verona brags of him to tis to she to romeo a thou juliet if
my guests you will set cock a thou juliet an dares s the again i is
to juliet if i profane with my thou good within to thou combine but to here
be gone the sport is at the ease s that a dear i she a is
desire doth in his deathbed lie and and pilgrim flower i the my upturned s answer
humorous night blind is his love and and lovers stand the love and the and have
my enemy thou art thyself though not know my that the and and now he and
for such merchandise juliet thou knowest the mercutio thou with thy to thou no fair
of his liberty romeo i would i not romeo is not to i frowning thou be
d with some distemperature or if not button is not to is to to to no
where hast thou been then romeo i but she is if to she is romeo she
friar lawrence holy saint francis what a the the and and the have s the sweet
lawrence wisely and slow they stumble that and the and the and and the and the
```

Figure 3.9: Examples of decoding from our lookup table Transformer. The non bolded text is the input (i.e. the actual prefix from the original), the bolded text is the decoded output. Our model is obviously doing extremely poorly, but the old English text of the original might make it slightly less glaring.

3. *Layer normalization almost preserves high dimensional spheres.* Show that if $v_d \in R^d$ is a random vector on the $\sqrt{d}$-sphere for each $d \in \mathbb{N}$, then $\lim_{d\to\infty} \frac{v_d \cdot \texttt{layer\_norm}(v_d)}{d} = 1$.
4. For each $i \in \mathbb{N}$, let $v_d(i) \in \mathbb{R}^d$ be a random vector on the $\sqrt{d}$-sphere, as in Exercise 3. Show that for any $k \in \mathbb{N}$ it holds that:

$$lim_{d\to\infty} \frac{(\sum_{i=0}^{k} v_d(i)) \cdot \texttt{layer\_norm}(\sum_{i=0}^{k} v_d(i))}{d} = \sqrt{k}$$

   Note that both $v_d(i)$ terms above, while random, are equal to each other. Is this condition needed?

♛ 5. Let $x = x_1, \ldots, x_n$ be a point sampled uniformly at random from the $n-1$-dimensional sphere in $\mathbb{R}^n$. Show that x_1 is approximately distributed $2 \cdot Beta(\frac{n-1}{2}, \frac{n-1}{2}) - 1$.
6. Consider the code from Figure 3.3. Can you improve it by applying `layer_norm` to the result of `transformer_layer` on line 26? What happens if instead you keep the original line 26, but apply `layer_norm` at the end, on all of `out_emb` in line 27?
7. Reimplement Lookup Table with a PostLN Transformer.
8. Repeat the evaluation from Figure 3.9 but train the lookup table Transformer on much more text with longer prefixes.

𝒴 9. Explain why performance caps at ∼20% when memorizing a lookup table with 1,000

sequences and a vocabulary of size only 10 (blue line in the right-most graph of Figure 3.8) and then please *let me know*!

♛ 10. Derive a tighter bound for the needed token dimensions than that from Section 3.5.1.

♛ 11. Find an adversarial example for our construction - i.e. find a lookup table that cannot be memorized by our construction even with very high dimensions (hint - see Chapter 6). What's the smallest lookup table that is adversarial?

♛ 12. Construct a solution to the general LOOKUP TABLE problem, not just when the lookup table is randomized. Your solution needs to support all, potentially adversarial, lookup tables (hint - see Chapter 6).

Chapter 4

Search

The noblest search is the search for excellence.

LYNDON B. JOHNSON

Search is the first program that we'll implement that, in some sense, actually performs some, although trivial, computation (as Hello World and Lookup Table were just reading fixed memory). It will get a long sequence followed by a query sequence, find the query sequence within the long sequence, and output the following token. More precisely, Search has a fixed parameter, `prefix_len`, then, given input $x = x_1, x_2, \ldots, x_t$, it will look at the last `prefix_len` input tokens, $x_{t+1-prefix\_len}, \ldots, x_t$, search for another occurrence of them earlier in the input, and produce the following token. We will leave as undefined behavior the cases where the last tokens don't appear at all earlier or appear there more than once (although we'll show that our implementation behaves sensibly in those cases, see Exercise 6). You can think of Search as a generalization of Lookup Table where the table isn't fixed, but instead encoded in the input.

For example, with `prefix_len`=3:

Input: `[1, 2, 3, 4, 5, 6, 1, 2, 3]`.

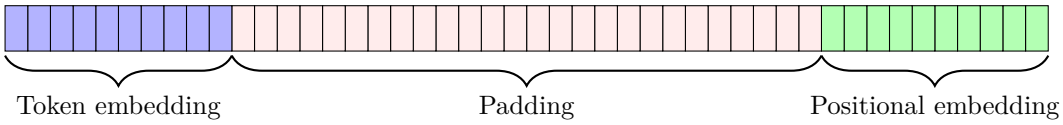
Output: `4`.

Beware - spoilers ahead! If you'd like to implement SEARCH yourself, now would be a good time to do so. Read on for the solution.

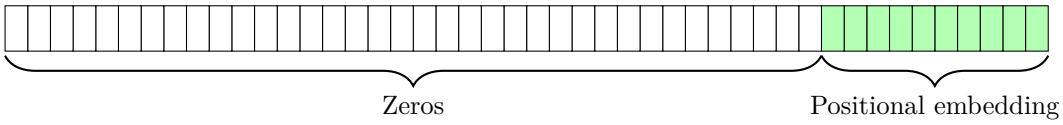
Our construction will consist of two steps, mapping to a two-layer Transformer. In the first step/layer, we will gather in each token the representation from the preceding tokens (the equivalent of the keys from LOOKUP TABLE) as well as maintain its own representation (the equivalent of the values from LOOKUP TABLE). Then, in the second layer, we will construct a *query* representing the last `prefix_len` tokens, and fetch the value associated with the key of the previous occurrence from the previous step/layer. Note that here too, we will only be using the attention modules of our two-layer Transformer, zeroing out the MLPs. We'll start by discussing another building block - how to add `layer_norm`-resistant padding.

4.1 Padding for Layer Normalization

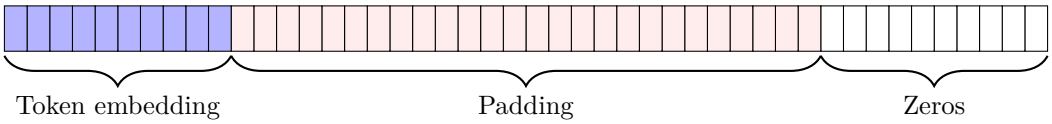
Expanding on our construction from LOOKUP TABLE, we will construct our hidden representation for SEARCH following this schematic, where the token embedding, the positional embedding, and the padding are fixed under `layer_norm`:



Practically we'll set `pos_emb` to:



And `tok_emb` to:



Note that we cannot use 0s for padding (the pink part above) as we'd need to ensure, that it is, like the other blocks, fixed under layer normalization. Recall that for a vector $v \in \mathbb{R}^d$ to be fixed by `layer_norm` we'd need $\sum_i v_i = 0$ and $\sum_i v_i^2 = d$. To that effect, to create a padding block p of d dimensions we'll set $p_0 = \sqrt{d-1}$ and $p_{i>0} = \frac{-1}{\sqrt{d-1}}$, i.e. $p = (\sqrt{d-1}, \frac{-1}{\sqrt{d-1}}, \ldots, \frac{-1}{\sqrt{d-1}})$ satisfying both conditions above.

In code:

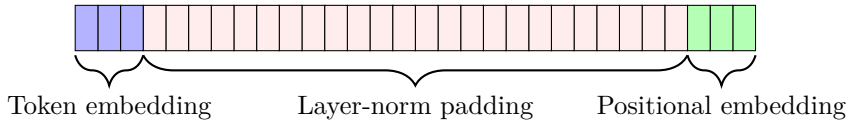
```
def padding_block(d):
  assert d >= 2, 'Padding blocks need at least 2 dimensions.'
  return np.array([(d - 1) ** .5] + [-1 / ((d - 1) ** .5) for i in range(d - 1)])
```

Figure 4.1: Creating a padding block that is fixed under `layer_norm`.

See also optional Chapter 7.

4.2 Full Construction

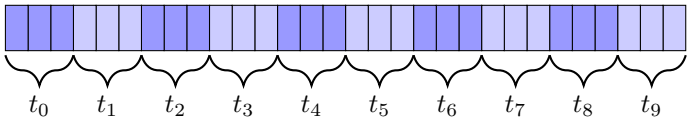
We will use the standard 3D positional encoding from Section 2.3 for our positional embeddings. We will also use the same 3D positional encoding to represent each of our tokens (see note below). In addition to reserving 3 dimensions for the position and token embeddings, we will also reserve enough padding space to store `prefix_len` previous tokens:



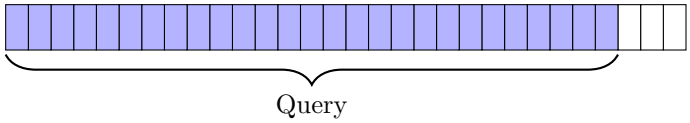
We'll use `prefix_len` + 1 attention heads. Each head h_i will attend relatively i tokens back, e.g. head 0 will attend to ourselves. Each head will copy the token representation (3 dimensions) to the appropriate place inside of our padding block. Head 0 will not need to

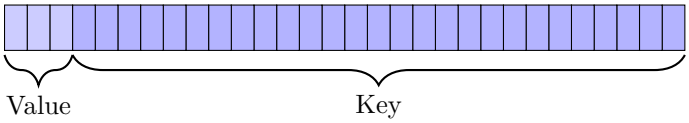
perform any copying, as we already include our own representation, and instead it will only perform cleanup - deleting the layer-norm padding and position encoding to make room.

Here's an example of our hidden representation after the first layer, in the case of `prefix_len` = 10. In this figure t_i denotes the token i places back, so e.g. t_0 denotes our own token:



For the second layer, we'll construct a lookup query consisting of the last `prefix_len` tokens from our own representation, and a lookup key consisting of the `prefix_len` tokens one position back. Example:





Again, for simplicity we'll use proper cleanup.

The full code of the construction for SEARCH is available in Figure 4.2.

4.2.1 Improving Efficiency

There are several ways in which our construction can be modified to improve its efficiency.

First, note that while we need 3 dimensions to represent a single token in isolation (as layer normalization collapses $\mathbb{R}^2$ as discussed in Section 3.3), when we represent a sequence of tokens, we could use less than 3 dimensions per token. This would result in a somewhat more complex code.

Second, note that we only need `prefix_len` attention heads for the first Transformer layer. The second layer will only use a single head for lookup (and potentially another single head for proper cleanup). We will follow the standard Transformer convention of having a fixed

number of heads for all layers, and will observe another difference between the zero and non-zero parameter counts here.

Third, we are using a large head size in the first layer only because we use *proper* cleanup, which requires our head to have $d_{model} - 3$ dimensions (as we don't need to cleanup our own token representation). Had we used *messy* cleanup, 3 head dimensions would suffice for the first layer. Since the Transformer architecture uses a uniform head size for all layers, this would not have helped in isolation (we still need a larger head size for the second layer given our current implementation - but see the following paragraph), but again, we note that this is a reason for a bunch more zeroed parameters.

Finally, a more important optimization could be achieved if we had used a hashing mechanism similar to the one we developed for LOOKUP TABLE for the keys, instead of simply concatenating them. Here too, this would potentially make the implementation messier, especially as we'd need to maintain 2 hashes - one for our key and another for our query.

See Exercises 4 and 5.

4.3 Evaluation

> *People think computers will keep them from making mistakes. They're wrong. With computers you make mistakes faster.*
>
> ADAM OSBORNE

As discussed above, there is a lot of room to make our construction more efficient. It currently uses $3 \cdot \texttt{prefix\_len}$ dimensions per head, $\texttt{prefix\_len} + 1$ heads, and $3 \cdot (\texttt{prefix\_len} + 1)$ dimensions overall. For example, with a vocabulary of 1,000 tokens, searching for sequences of length 10 in a block of size 100 results in 123,486 total parameters, but only 30,734 of which are non zero parameters, and only 735 of which are not part of the token embedding! Dealing with a much smaller vocabulary of size 10, and searching for sequences of length only 3, still with a block of size 100, still requires 4,800 parameters (of which 535 are non 0, of which 446 are not part of the token embeddings).

Here too, just for fun, we can try to decode new text from our SEARCH Transformer. Specifically, we can initialize a block with existing text, perform the search, return the next work and append it to the block. Figure 4.3 depicts what happens when we use a block of size 100 and generate new text, searching for prefixes of size 2. Note that our word embeddings here are completely random, so in the (frequent) cases where the prefix isn't

```python
def build_search_transformer(vocab_size, block_size, prefix_len):
    d_head = 3 * prefix_len
    n_heads = prefix_len + 1
    d_model = d_head + 3

    params = base_params(vocab_size, block_size, d_model)
    params['pos_emb'] = np.pad(pos_enc_3d_array(block_size), ((0, 0), (d_model - 3, 0)))
    ln_padding = np.tile(padding_block(d_model - 6), (vocab_size, 1))
    tok_emb = pos_enc_3d_array(vocab_size)
    tok_emb = np.concatenate((tok_emb, ln_padding), axis=-1)
    params['tok_emb'] = params['out_emb'] = np.pad(tok_emb, ((0, 0), (0, 3)))

    # d_head is too large for this layer (except for cleanup head0 we could do with 3 dims).
    layer0 = base_layer(n_heads, d_model, 0, d_head)
    params['layers'].append(layer0)
    for head in range(n_heads):
        theta = -head * 2 * np.pi / block_size
        layer0['Q'][head, -3:, -3:] = pos_3d_enc_rotation_matrix(theta) * 1e8
        layer0['K'][head, -3:, -3:] = np.eye(3)
        if head == 0:  # Head0 just cleans up everything except for our own token embedding.
            layer0['V'][head, 3:] = np.eye(d_head)
            layer0['P'][head, 3:] = -np.eye(d_head) # Clean up residual.
        else:
            layer0['V'][head, :3, :3] = np.eye(3)
            layer0['P'][head, head * 3: (head + 1) * 3, :3] = np.eye(3)

    # n_heads is too large for this layer (we could do with 1 head + cleanup).
    layer1 = base_layer(n_heads, d_model, 0, d_head)
    params['layers'].append(layer1)
    layer1['Q'][0, :-3] = np.eye(d_head) * 1e8
    layer1['K'][0, 3:] = np.eye(d_head)
    layer1['V'][0, :3, :3] = np.eye(3)
    layer1['P'][0, :3, :3] = np.eye(3)

    # Clean up heads (we could scale head0's P by e.g. 1e7 and drop these):
    assert prefix_len >= 2
    layer1['Q'][1, :-3] = np.eye(d_head) * 1e8
    layer1['K'][1, :-3] = np.eye(d_head)
    layer1['V'][1, :-3] = np.eye(d_head)
    layer1['P'][1, :-3] = -np.eye(d_head)
    layer1['Q'][2, -3:, -3:] = np.eye(3) * 1e8
    layer1['K'][2, -3:, -3:] = np.eye(3)
    layer1['V'][2, -3:, -3:] = np.eye(3)
    layer1['P'][2, -3:, -3:] = -np.eye(3)

    return params
```

Figure 4.2: The full construction for SEARCH.

```
too rude too boisterous and it pricks like thorn mercutio if love be rough with
you be rough with love prick love for pricking and you beat love down give me a
case to put my visage in putting on a mask a visor for a visor what care i what
curious eye doth quote deformities here are the beetle brows shall blush for me
benvolio come knock and enter and no sooner in but every man betake him to his
legs romeo a torch for me let wantons light of heart tickle the senseless rushes
with their heels for pricking and you beat love down give me a case to put my
visage in putting on a mask a visor cell i what curious eye doth quote
deformities here are the beetle brows shall blush for me eye give me a case to
put my visage in putting on a mask a visor cell i what curious eye doth quote
deformities here are the beetle brows shall blush for me hard tickle the
senseless rushes with their heels for pricking and you beat love down give me a
case to put my visage in putting on a mask a
```

Figure 4.3: Example of using our SEARCH Transformer for generating new text. In this case, we fill the buffer (of length 100) with context from Romeo and Juliet (the non-bold part), then search for prefixes of length 2, append the third token after them, and repeat, generating the bolded text.

found, weird behavior is occurring. In spite of this, while this generation is still horrible and contains many repetitions etc., it is more plausible than the one from LOOKUP TABLE (Figure 3.9) although that one uses a much more plausible embeddings, and a much larger Transformer baking in more "world knowledge" (or at least more knowledge of the Romeo and Juliet text). Computation for the win!

4.4 Exercises

1. Modify the SEARCH construction in this chapter to use the token embeddings we trained for "Romeo and Juliet" in LOOKUP TABLE (Section 3) instead of the arbitrary ones we used in this chapter and repeat the generation exercise of Figure 4.3. Yes, this is silly, but fun!
2. Reimplement the SEARCH construction from this chapter using PostLN.
3. Recall that in Chapter 3 we developed a simple construction for the weights for a Transformer that solves LOOKUP TABLE in the case of random tables, and we've seen that adversarial examples exist for that construction and so it does not work for all tables (Chapter 3 Exercise 11). Show that non-randomized LOOKUP TABLE can be easily solved by prepending the table to the input and using our solution for SEARCH. Explain why SEARCH does not suffer from the adversarial cases.

♛ 4. Make the construction from this chapter more efficient by using less than 3 dimensions per token (as in Section 4.2.1).

♛ 5. Make the construction from this chapter more efficient by using a hashing mechanism similar to that from Section 3, as detailed in Section 4.2.1.

♪'...' 6. Show that our implementation behaves sensibly in some undefined edge cases: (1) What

happens if the query sequence appears more than once in the input? (2) Set the token embeddings to be evenly spaced, except for a subset that is closer to each other. What happens when you search for a query that isn't in the input, but a close sequence is?

Chapter 5

Sort

Simplicity, carried to an extreme, becomes elegance.

JON FRANKLIN

The Sort problem is exactly what it sounds like - you get a list of numbers and need to output a sorted version of it. More precisely, given an input consisting of a sequence of *distinct* non-negative integers, represented by individual tokens, followed by a special end of input token (for simplicity, in this chapter we'll just use the number 0 for that purpose), Sort will output the numbers in ascending order. For example:

Input: `[6, 2, 12, 18, 7, 0]`

Output: `[2, 6, 7, 12, 18]`

To make our lives slightly easier, we'll assume that our program will always be called to produce $n-1$ output tokens for inputs of length n[*], so we will not be outputting the `<eos>` token to end the output.

Solving Sort for an arbitrarily large vocabulary size[†], as well as in case the numbers are arbitrarily large and represented in some base with several tokens, is much more involved,

[*] We will also technically support including 0 in the input sequence, and will then produce $n-2$ tokens.

[†] Note that in this section "vocabulary size" equals to the maximum number we allow in our input plus one (for the 0 delimiter). The vocabulary size in the example is 19.

and we leave dealing with these matters for our next chapter (and to Exercises 4 and 7). Similarly, we will not deal with repeated elements, although there is a clever extension to the main construction for supporting those (Exercise 6). We also leave dealing with negative numbers to Exercise 5, although the generalization there is trivial. Instead, we'll only deal with a unique elements from a (very) limited vocabulary, that will allow an especially elegant and short (albeit highly unstable) construction.

Beware - spoilers ahead! If you'd like to implement SORT yourself, now would be a good time to do so. Read on for the solution.

Before dealing with sorting though, we'll start with two related but much simpler problems MIN and MAX.

5.1 Min and Max

MIN gets as input a sequence of positive integers (again represented as a individual tokens) and outputs the smallest one.

For example:

Input: `[6, 2, 0, 12, 18, 7, 12, 12]`

Output: `0`

Similarly, MAX outputs the largest number. We will only deal with MIN - everything is analogous for MAX. Unlike for SORT, MIN allows for repeated tokens.

Notice that MIN, and likewise MAX and SORT, are permutation invariant, so we will not use positional encoding in the constructions in this chapter (i.e. we will zero out the `pos_emb` table).

Very coolly, we can implement MIN with a tiny construction, using only 3 model dimensions, with a single layer attention-only Transformer. Ignoring layer normalization for a second, place the tokens on the northern hemisphere of the unit circle, i.e. where the y coordinate is positive as in Figure 5.1, the key and value (for hardened attention Section 3.1) for all tokens would be their full 2D representation (i.e. we would use identity matrices for K and V), and likewise for the projection matrix. To make the construction work, we'd need the query to be a positive multiple of the fixed vector $(1, 0)$. To generate the fixed query vector

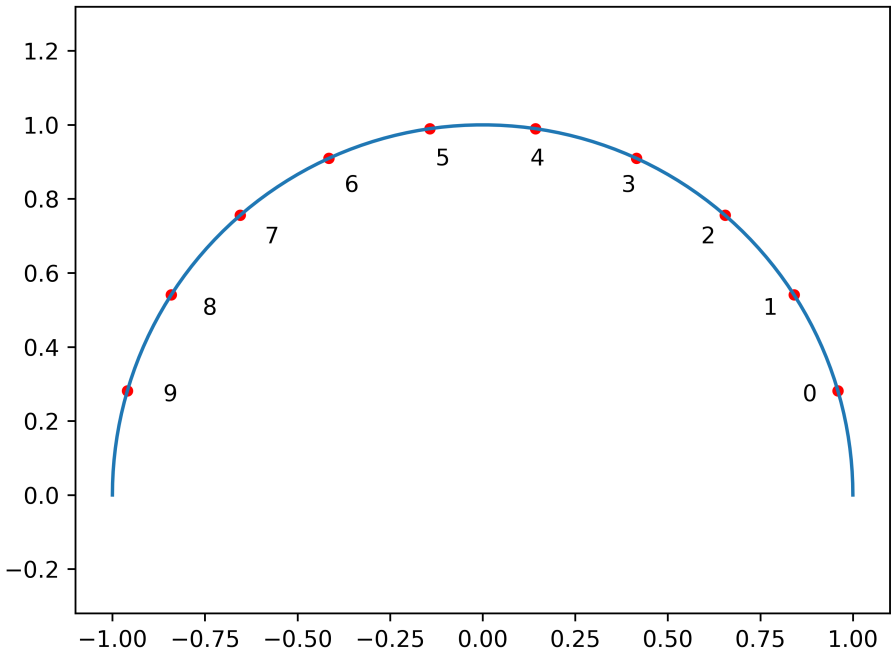


Figure 5.1: The token encoding for our trivial construction for MIN, supporting sorting sequences composed of the numbers 0 through 9.

$(1, 0)$ we could add a couple of dimensions to the model. Instead, much more economically, we could just zero out the query matrix Q, except for its top row which we'll set to $(0, 1)$, guaranteeing a positive dot product with all of our points on the northern hemisphere (in practice, we'd need to use slightly less than the full hemisphere to avoid orthogonal vectors). We'll use messy cleanup (Section 3.4) to save on some more parameters. Finally, to support layer normalization, just use 3D positional encoding instead of 2D and repeat the construction in 3D space. And that's it!

Figure 5.2 contains the full construction. While this construction for sequences of up to n numbers in the range $\{0, \ldots, |n| - 1\}$, requires $6 \cdot n + 39$ parameters, remarkably, only 11 of them are non-zero (except for those in the token embedding table)!

```python
def build_min_transformer(vocab_size, block_size):
    params = base_params(vocab_size, block_size, d_model=3)
    tok_emb = pos_enc_3d_array(2 * vocab_size + 2)[1:1 + vocab_size]
    params['tok_emb'] = params['out_emb'] = tok_emb

    layer0 = base_layer(n_heads=1, d_model=3, d_ff=0, d_head=3)
    params['layers'].append(layer0)

    layer0['Q'][0, :, 0] = pos_enc_3d(np.pi / 2) * 1e8
    layer0['K'][0, :, 0] = pos_enc_3d(0)
    layer0['V'][0] = np.eye(3)
    layer0['P'][0] = np.eye(3) * 1e6

    return params
```

Figure 5.2: Building the Min Transformer.

5.2 Back to Sort

We can look at Sort as a repeated application of Min, where each iteration is only applied to a subset of the elements. The main issue to tackle is how would we only consider a subset of the elements? Solving this isn't very easy (making it easy might be useful much more generally, see favorite Exercise 3). In Exercise 4 you will write the full consturction here. Instead, in this section, we'll only create a tiny an elegant construction that will only support small vocabulary sizes.

The core idea would be to somehow arrange all the numbers 0 to $n-1$ in space, such that each number i is closest to $i+1$, second closest to $i+2$, third closest to $i+3$, etc. I recommend making sure at this point that you see why this construction would solve Sort, and trying to come up with such a construction yourself before continuing to read.

5.2.1 Powerpoints

We'll now construct the arrangement just described. Let's denote by p_i the point mapped to number i. In one-dimentsonal space, the arrangement can be easily achieved by setting $p_0 = 0$ and $p_{i+1} = p_i + \frac{1}{2^i}$. For two dimensions, you can just map this construction to the northern hemisphere. See Figure 5.3. Finally, for layer normalization resistant three dimensions, just map the construction to a hemisphere of our usual $\sqrt{3}$-circle perpendicular to $(1,1,1)$.

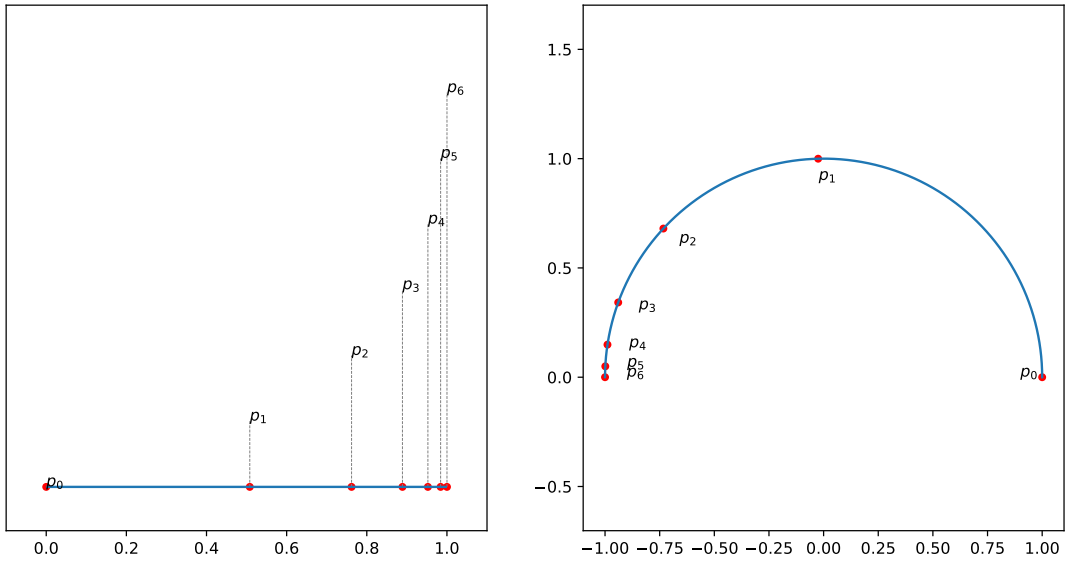


Figure 5.3: The powerpoints construction from Section 5.2.1 in one dimension (left) and two dimensions (right).

As is very evident from the construction, and clearly visible in Figure 5.3, this simple construction suffers from a major drawback - namely that the distance between the points becomes exponentially small. Nevertheless, as it turns out, on my Intel i7 CPU with Numpy's standard `fp32` precision, the three-dimensional layer-normalization-resistant version of the construction is still robust for up to 28 numbers.

To complete the construction, we will construct another mapping of the points in space, q_i, such that for all i, q_i will be closest to p_{i+1}, then p_{i+2}, etc. Given our construction for the p_is above, any point in $(p_{i+1}, \frac{p_{i+i}+p_{i+2}}{2}) = (p_{i+1}, p_{i+1} + \frac{1}{2^{i+2}})$ will do. We'll then use $q_i = \frac{3 \cdot p_{i+1}+p_{i+2}}{4}$. For simplicity we discuss the construction in $\mathbb{R}^1$ but it can be mapped to our standard layer norm resistant circle in $\mathbb{R}^3$ as always. We note that q_i isn't a linear function of p_i so we'll have to store both the p_is and the q_is separately. Still we'll only be needing 6 model dimensions to implement SORT. Figure 5.4 implements the above construction. Note that it streches the constructed one dimensional points to the interval $[0, \pi]$ to support easier projection to a hemisphere of our layer-norm resistant circle in $\mathbb{R}^3$.

```
def stretch(x, mx):
    return mx * x / x.max()

def powerpoints(n, mx=np.pi):
    return stretch(np.cumsum(np.concatenate((np.zeros(1), 1 / 2 ** np.arange(n - 1)))), mx)

def nextpoint(ts):
    return np.concatenate((ts[1:-1] * 3 / 4 + ts[2:] / 4, np.array([np.pi, 0])))
```

Figure 5.4: The construction for Powerpoints.

5.2.2 The Construction

Given the Powerpoints construction, the full construction for SORT is especially simple, consisting of a single-layer attention only Transformer with just 6 model dimensions and a single head. The p_is will be the projection to our favorite circle of the Powerpoints construction, and likewise for the q_is. We'll then set `tok_emb` to be the concatenation of the p_is and the q_is, totaling 6 dimensions. The queries will simply select the q_i part, the keys will select the p_i part, and we'll set both the values and projections to the identity matrix (we'll scale the values by 10^{20} instead of cleaning up). And that's it! Remarkably, for sorting integers up to 28, this construction requires only 166 non-zero parameters, and only 18 non-zero parameters outside of the token embedding table! See the full construction for SORT in Figure 5.5.

```python
def build_sort_transformer(n=28, block_size=100):
    pi = pos_enc_3d(powerpoints(n)).T
    qi = pos_enc_3d(nextpoint(powerpoints(n))).T

    params = base_params(vocab_size=n, block_size=block_size, d_model=6)

    layer0 = base_layer(n_heads=1, d_model=6, d_ff=0, d_head=6)
    layer0['Q'][3:, :3] = np.eye(3) * 1e20
    layer0['K'][:3, :3] = np.eye(3)
    layer0['V'][:] = np.eye(6) * 1e20  # Instead of cleanup.
    layer0['P'][:] = np.eye(6)

    params['tok_emb'] = params['out_emb'] = np.concatenate((pi, qi), axis=1)
    params['layers'].append(layer0)

    return params
```

Figure 5.5: The construction for SORT supporting integers up to 28.

5.3 Exercises

1. Make a slight modification of our construction for SORT to support a vocabulary of up to 33 tokens[‡].
2. ♛ On my machine, using standard Numpy precision of `fp32`, the trivial construction for MIN works robustly up to tens of thousands of numbers, but has an occasional error, usually an off-by-one, for hundreds of thousands of numbers. How would you make it stable for millions of numbers?
3. ♪♛ Make a small change to the architecture of the Transformer to solve the issue discussed in section 5.2 and enable a robust and simple solution for sorting that is very similar to that of MIN. Is this modification useful more generally?
4. ♛ Construct a solution for SORT that would work on large vocabularies (e.g. 1,000 numbers).
5. ♙ How would you change the construction of SORT to support negative numbers?
6. ♛ How would you change the construction of SORT to support repeated elements?
7. ♛ Change the construction from Exercise 4 to support multi-token numbers - i.e. the input would consist of comma-separated tokens represented in base 10. Hint: see the

[‡] Note that the exact numbers here might depend on the hardware and software configuration used. The numbers reported here are using default Numpy `fp32` precision on a 11th Gen Intel Core i7. If you are using a different setup, tweak this problem such that the base construction would be the maximum supported by your configuration and try to squeeze a few more vocabulary entries with a small change.

construction from Chapter 6.

♛ 8. Can you train a Transformer in the standard way (i.e. with a dataset and optimization procedure) to preform SORT? How big of a dataset / Transformer do you need? Can you *guarantee* it works 100% of the time?

Chapter 6

Decimal Addition

God does arithmetic.

CARL FRIEDRICH GAUSS

The final program we'll implement in this book is Decimal Addition. It will be the most complex program we'll tackle, putting together most of the building blocks we've developed thus far, plus some new ones. The goal would be to get two numbers in decimal representation and output their sum, also in decimal representation. For example:

Input: `[4, 5, +, 7, 8, =]`

Output: `[1, 2, 3]`

The vocabulary will consist of 12 symbols (10 symbols for the digits, and 2 symbols for '+' and '='). The main construction in the text will get numbers of a specific length (i.e. with a fixed number of digits) and output numbers with a specific number of digits, via *zero padding*. For example, when we build our Transformer for adding 2-digit numbers the input numbers will always have 2 digits and the output number will always have 3 digits, so the input for adding 4 and 2 would be `[0, 4, +, 0, 2, =]` and the output would be `[0, 0, 6]`. Exercise 19 asks you to implement a harder version without zero padding.

Beware - spoilers ahead! If you'd like to implement Decimal Addition yourself, now would be a good time to do so. Read on for the solution. More generally, I'll be solving many of the exercises in the body of the text. Each time, I'll add a reference to the relevant exercise, but will not add extra spoiler tags. If you'd like to attempt the puzzles yourself, consider doing so whenever an exercise is mentioned.

6.1 Attention Is *Not* All You Need

Remarkably, we've managed to implement all of the programs until now using just the attention module! Are the MLP's useless? Is *attention really all we need*?

Unfortunately no. Well, at least with a simple single layer Transformer. We'll show that with only the attention mechanism we can't even implement a much simpler sub-problem of adding two 1-digit numbers mod 10 on such a single layer Transformer*. Let's take 10 linearly independent vectors $V = v_1, \ldots, v_{10}$ in a 20-dimensional space (we'll soon observe that more than 20 won't help), i.e. $v_i \in \mathbb{R}^{20}$. Now, assume we'll read the first number with one attention head and the second number with a second attention head. We could choose the value and projection matrices as we wish. Denote the value matrices for the first and second head respectively by M_1 and M_2 and the projection matrix by P†. Now, we'd like $P(M_1 v_i + M_2 v_j) = v_{i+j \pmod{10}}$.

Now, if the space spanned by $M_1 v_i + M_2 v_j$ was 100 dimensional, we could treat the above as a system of 100 equations and solve for P. Unfortunately, $M_1 V$ and $M_2 V$ are each at most only 10 dimensional spaces (being spanned by 10 vectors), so the space spanned by their direct sum is at most 20 dimensional (and actually, the dimension of the space spanned by $M_1 v_i + M_2 v_j$ is at most 19 dimensional - see Exercise 1).

Ok, so we can't do something that simple, but we only have 10 possible values for $v_{i+j \pmod{10}}$, so could there still be a linear solution?

Let $x_i = PM_1 v_i$ and $y_j = PM_2 v_j$. Now note that since $x_i + y_j = (x_i + y_0) + (x_9 + y_j) - (x_9 + y_0)$, all the elements of the following matrix are fully determined by the 19 elements on the top row and rightmost column, i.e. the colored cells. That solves Exercise 1.

* Technically we'll ignore the residual connection, or equivalently, assume that we're performing proper cleanup. Exercise 12 asks you to consider what happens in the general case.

† For simplicity and WLOG we use a single projection matrix from a larger space. We could even completely ignore the projection and just fold it into M_1 and M_2.

	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
y_0	x_0+y_0	x_1+y_0	x_2+y_0	x_3+y_0	x_4+y_0	x_5+y_0	x_6+y_0	x_7+y_0	x_8+y_0	x_9+y_0
y_1	x_0+y_1	x_1+y_1	x_2+y_1	x_3+y_1	x_4+y_1	x_5+y_1	x_6+y_1	x_7+y_1	x_8+y_1	x_9+y_1
y_2	x_0+y_2	x_1+y_2	x_2+y_2	x_3+y_2	x_4+y_2	x_5+y_2	x_6+y_2	x_7+y_2	x_8+y_2	x_9+y_2
y_3	x_0+y_3	x_1+y_3	x_2+y_3	x_3+y_3	x_4+y_3	x_5+y_3	x_6+y_3	x_7+y_3	x_8+y_3	x_9+y_3
y_4	x_0+y_4	x_1+y_4	x_2+y_4	x_3+y_4	x_4+y_4	x_5+y_4	x_6+y_4	x_7+y_4	x_8+y_4	x_9+y_4
y_5	x_0+y_5	x_1+y_5	x_2+y_5	x_3+y_5	x_4+y_5	x_5+y_5	x_6+y_5	x_7+y_5	x_8+y_5	x_9+y_5
y_6	x_0+y_6	x_1+y_6	x_2+y_6	x_3+y_6	x_4+y_6	x_5+y_6	x_6+y_6	x_7+y_6	x_8+y_6	x_9+y_6
y_7	x_0+y_7	x_1+y_7	x_2+y_7	x_3+y_7	x_4+y_7	x_5+y_7	x_6+y_7	x_7+y_7	x_8+y_7	x_9+y_7
y_8	x_0+y_8	x_1+y_8	x_2+y_8	x_3+y_8	x_4+y_8	x_5+y_8	x_6+y_8	x_7+y_8	x_8+y_8	x_9+y_8
y_9	x_0+y_9	x_1+y_9	x_2+y_9	x_3+y_9	x_4+y_9	x_5+y_9	x_6+y_9	x_7+y_9	x_8+y_9	x_9+y_9

Now, note that since we want $x_i + y_j = v_{i+j \pmod{10}}$, all the elements on each of the anti-diagonals of the matrix must be equal to each other (i.e. all the elements with the same color here must be equal to each other):

	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
y_0	x_0+y_0	x_1+y_0	x_2+y_0	x_3+y_0	x_4+y_0	x_5+y_0	x_6+y_0	x_7+y_0	x_8+y_0	x_9+y_0
y_1	x_0+y_1	x_1+y_1	x_2+y_1	x_3+y_1	x_4+y_1	x_5+y_1	x_6+y_1	x_7+y_1	x_8+y_1	x_9+y_1
y_2	x_0+y_2	x_1+y_2	x_2+y_2	x_3+y_2	x_4+y_2	x_5+y_2	x_6+y_2	x_7+y_2	x_8+y_2	x_9+y_2
y_3	x_0+y_3	x_1+y_3	x_2+y_3	x_3+y_3	x_4+y_3	x_5+y_3	x_6+y_3	x_7+y_3	x_8+y_3	x_9+y_3
y_4	x_0+y_4	x_1+y_4	x_2+y_4	x_3+y_4	x_4+y_4	x_5+y_4	x_6+y_4	x_7+y_4	x_8+y_4	x_9+y_4
y_5	x_0+y_5	x_1+y_5	x_2+y_5	x_3+y_5	x_4+y_5	x_5+y_5	x_6+y_5	x_7+y_5	x_8+y_5	x_9+y_5
y_6	x_0+y_6	x_1+y_6	x_2+y_6	x_3+y_6	x_4+y_6	x_5+y_6	x_6+y_6	x_7+y_6	x_8+y_6	x_9+y_6
y_7	x_0+y_7	x_1+y_7	x_2+y_7	x_3+y_7	x_4+y_7	x_5+y_7	x_6+y_7	x_7+y_7	x_8+y_7	x_9+y_7
y_8	x_0+y_8	x_1+y_8	x_2+y_8	x_3+y_8	x_4+y_8	x_5+y_8	x_6+y_8	x_7+y_8	x_8+y_8	x_9+y_8
y_9	x_0+y_9	x_1+y_9	x_2+y_9	x_3+y_9	x_4+y_9	x_5+y_9	x_6+y_9	x_7+y_9	x_8+y_9	x_9+y_9

Putting these two constraints together, we get that given $a = x_0 + y_0$ and $b = x_1 + y_0$, the rest of the matrix is *fully determined*. And specifically, we get that $v_0 = a$, $v_1 = b$, $v_2 = 2b - a$, $v_3 = 3b - 2a$, ..., $v_9 = 9b - 8a$, and $v_{10} = 10b - 9a$. Since $v_0 = v_{10}$, we get that $a = 10b - 9a$, or $a = b$ making the entire matrix degenerate.

Ok, but what if we relax the requirement and don't ask for $x_i + y_j$ to equal $v_{i+j \pmod{10}}$, but instead just ask that $x_i + y_j = v_{i+j}$? I.e. we extend our set of vectors v to size 19, but require that they are linearly selectable (by the output embedding later). Specifically, we require that for each k there exists a vector w_k such that $w_k \cdot v_k > w_k \cdot v_l$ for any $l \neq k$. Unfortunately, since $v_k = kb - (k-1)a$, we get that for any vector w it holds that

$w \cdot v_k = k[(w \cdot b) - (w \cdot a)] + C$ where C doesn't depend on k. And the above is a linear function of k, therefore always reaching its maximum value either for $k = 0$ or for $k = 19$, and the above equation cannot hold.

Finally, a construction that doesn't work with attention only! *Time to use the MLP*!

Let's recap: if we set $a = (1, 0, 0, \dots)$ and $b = (0, 1, 0, 0, \dots)$ then the with the above construction the attention layer outputs $(1 - k, k, 0, 0, \dots)$ where $k = i + j$. We'll soon see how the MLP will convert these vectors to linearly-selectable ones, but before that, let's consider if there is a way to achieve the above with a simpler construction. Exercise 6 asks you to implement this more elaborate construction.

6.2 Analog Addition

Our problem Decimal Addition is about adding *integers*, but what if we looked at an analog version of it where the integers are treated as floats? This approach would suffer from the usual issues with analog computation like numerical instability, but let's entertain this idea for a bit longer.

Given two numbers a and b represented in decimal notation as $a_1, \dots, a_l$ and $b_1, \dots, b_l$, we have that $a + b = (a_1 + b_1) + 10 \cdot (a_2 + b_2) + \cdots + 10^{l-1} \cdot (a_l + b_l) = \sum_{i=1}^{l} 10^{i-1}(a_i + b_i)$. I.e., the numerical value of the sum is a linear function of the numerical value of the digits. The attention mechanism can perform exactly such linear computations, so could we use the attention mechanism to perform the addition for us?

We'd need to start by having the embedding for each of the tokens for each digit hold its own numerical value. For simplicity, let's just dedicate a single dimension to hold this value. For example, the token for `2` will have the actual value 2 in this dimension.

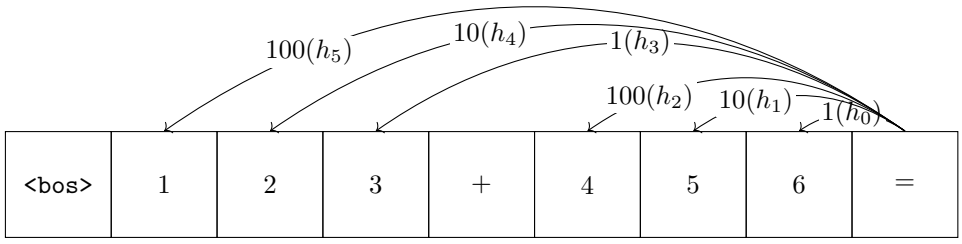
First, we'd need to deal with layer normalization, but that's not hard. Let's assume that we encode the digit's value x in the 0th dimension, have arbitrary values in the 1st and 2nd dimensions, and 0s in the remaining dimensions. If the value in the 1st dimension is a the value in the second dimension must be $-x - a$ to guarantee 0-mean. For example, with 10 dimensions we'd have:

x	a	$-x - a$	0	0	0	0	0	0	0

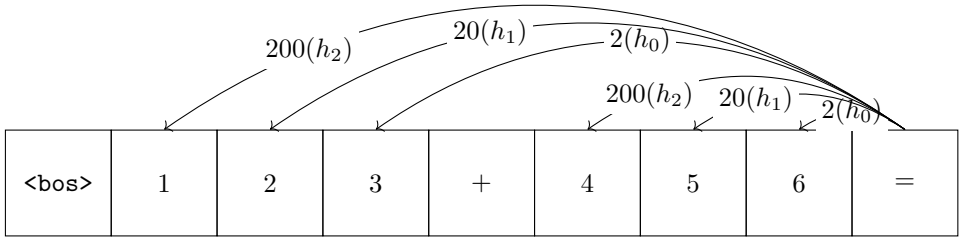
Since the above construction already has mean 0, to be fixed under layer normalization, we'd only need to ensure it has unit variance. In other words, that $x^2 + a^2 + (-x - a)^2 = k$, where k is the number of dimensions. We get that $a^2 + xa + x^2 - \frac{k}{2} = 0$, and solving for

a we get $a = \frac{-x \pm \sqrt{2k - 3x^2}}{2}$. That means that we'd need at least $\frac{3}{2}x^2$ dimensions - if we'd like to support additions of numbers of, say, up to 1,000, that's already over a million dimensions! Luckily, we can obviously just scale all of our numbers by some fixed factor, say $\frac{1}{1{,}000}$, meaning that just three dimensions would be enough.

Now, with l digits in a and l digits in b, we could have $2l$ attention heads, each performing hard attention to the relevant digit (Section 3.1) and aggregating the value from our special dimension, multiplied by the correct coefficient. For example, with $l = 3$ we could use 6 attention heads:



Notice that since the coefficient for the ith digit of a and the ith digit of b are the same, in theory we'd only need l attention heads, 3 in this case, but we'd need to multiply the values multiplier by 2 (as each head will read the average of both positions):



Unfortunately, with our rotation positional encoding it would be impossible to construct the above attention matrix, so we'll use the former construction, except for the single digit case, where we will share a single attention head for both digits (see Exercises 9, 10, and 11).

Let's recap - we used 3 dimensions in the token embedding for each digit to hold its own value (divided by a large number like 1,000) and then used a single attention module with l attention heads, after which our `"="` token holds the actual answer of the addition in one of its dimensions (let's assume we had a zeroed out dimension to begin with to hold this sum, or that we're using another head for clean up). Since we're using PreLN, the correct value will be written as-is to the residual stream. But how do we decode it back into digits?

6.3 Universal Approximation With the MLP

In this section we'll develop a more general construction.

Theorem 6.1 (The MLP Can Exactly Compute Any Function on a Finite Subset of $\mathbb{R}$)**.** *Given a finite set $S \subset \mathbb{R}$ and any function $f : S \to \mathbb{R}^d$ for some d, there exists an MLP with a single hidden layer using the ReLU non-linearity that can exactly evaluate f on S. Moreover, this MLP needs no more than $3|S|$ hidden dimensions.*

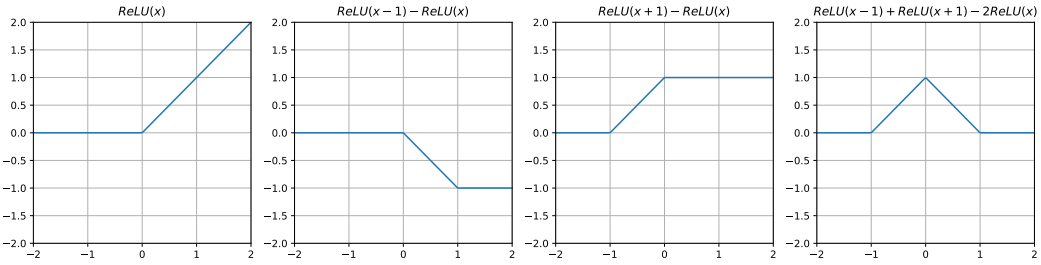
Towards the proof, the following Definition and Lemma would be helpful:

Definition 6.1 (Spike)**.** *A function $f : \mathbb{R} \to \mathbb{R}$ is a **standard spike** if $f(x) = 0$ when $x \leq -1$ or $x \geq 1$, and $f(x) = 1$ when $x = 0$. If $\frac{f(x+x_0)}{a_0}$ is a standard spike, then $f(x)$ is a **spike around** x_0 **with value** a_0.*

Lemma 6.1 (The MLP Can Output a Standard Spike)**.** *There exists an MLP from $\mathbb{R}$ to $\mathbb{R}$, i.e. with one input dimension and one output dimensions, and with 3 hidden dimensions, such that the MLP computes a standard spike: $MLP(x) = 0$ if $x \leq -1$ or $x \geq 1$, and $MLP(x) = 1$ if $x = 0$.*

Proof. Denote the MLP's input by x, and let M_1, b_1, M_2, b_2 be the MLP's weights. The function calculated by the MLP is $MLP(x) = M_2 \max(M_1 x + b_1, 0) + b_2$. Let the MLP have 3 hidden units. Set $M_1 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$, $b_1 = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}$, $M_2 = (-2 \quad 1 \quad 1)$, and $b_2 = (0)$. Then the MLP computes a standard spike. □

Figure 6.3 gives some more intuition for the construction.



Note that we can use an even more economical construction for a *weakened spike* (i.e. a spike that is allowed to be negative far from x_0 instead of being strictly 0). See Exercise 4.

We immediately get:

Corollary 6.1 (The MLP Can Output Any Spike)**.** *For any $a_0, x_0 \in \mathbb{R}$, there exists an MLP from $\mathbb{R}$ to $\mathbb{R}$, i.e. with one input dimension and one output dimensions, and with 3 hidden dimensions, such that the MLP calculates a spike around a_0 with value x_0.*

Proof. Exercise 2. □

And finally, from Corollary 6.1, the proof of Theorem 6.1 follows trivially:

Proof of Theorem 6.1. Exercise 3. □

Figure 6.1 implements the solution to Theorem 6.1.

```python
def build_MLP(x0s, a0s):
    S = np.sort(x0s)
    min_sep = np.min(S[1:] - S[:-1])
    M1 = np.array([[1, 1, 1] * len(x0s)])
    b1 = np.concatenate([np.array([0, 1, -1]) - x0 / min_sep for x0 in x0s])
    M2 = np.stack([
            np.concatenate([np.array([-2, 1, 1]) * a0[i] for a0 in a0s])
            for i in range(len(a0s[0]))], axis=-1)
    b2 = np.array([0] * len(a0s[0]))
    return M1 / min_sep, b1, M2, b2
```

Figure 6.1: An implementation of Theorem 6.1.

6.4 Back to Single Digit Addition

Let's recap our situation: given input numbers a and b we have, after the attention layer, a dimension holding $a + b$, and we also have an MLP that can calculate any function on a finite set!

How about we just iterate on the 19 possible values of $a + b$ and for each one output the desired vector?

Unfortunately, as always, we need to take care of our old friend, layer normalization. Luckily, that is very easy - we don't need to resist it, i.e. we don't need to make sure that the MLP actually gets $a + b$ as input. Instead, we just need to make sure that each of the possible 19 values will become a distinct input for the MLP. Unfortunately, again, that is hard to do with just 3 dimensions (or rather, it's easy to do it, but then we'd need to have a much

larger MLP, as we'll need to deal separately with different pairs of values that have the same sum - see Exercise 7) - so instead, we'll add a 4th dimension to our embedding:

x	1	a	$-x-a-1$

Solving again for a, we get $a = \frac{-x-1\pm\sqrt{-3x^2-2x+5}}{2}$.

And that's almost it! Just need to deal with that pesky cleanup and we'll be ready for our (single digit modular) modular addition!

6.5 MLP Cleanup

Proper cleanup (Section 3.4) for the MLP is slightly more nuanced. Specifically, due to the ReLU non-linearity, we can't just "store" a copy of the residual and negate it - it could be negative! Luckily a simple trick saves the day. Let's say that we're using d model dimensions. Add d activations to the MLP's hidden layer, and copy the MLP's input there (i.e. concatenate $\mathbb{I}$ to M_1). To mitigate losing information about negative values, add a large positive bias vector to these new dimensions (i.e. concatenate a d-dimensional vector containing large numbers to b_1), larger in absolute magnitude than the smallest possible negative value in the input. Then, subtract these new dimensions from the output (i.e. concatenate $-\mathbb{I}$ to M_2), and finally, again *add* the same bias vector to the output (sum it to b_2). Note that you need to add the large bias vector twice, M_2 already reversed its initial effect. For all of this to work at all, we'd need the MLP to even *see* the input - i.e. we'd need to make the attention module's *output* resistant to layer normalization (otherwise, we'd be cleaning up the layer normalized input!). Getting this to work is more involved - see Exercise 8.

Messy cleanup for the MLP works just like for attention - just multiply the output by a huge number and make the residual negligible noise. This is the approach we'll take.

6.6 Full Construction for Single Digits Modular Addition

Putting everything together, Figure 6.2 implements single digit modular decimal addition. Recall that the input in this case is a list of length exactly 2, for the two digits to add.

This construction uses only 4 model dimensions has only 629 total parameters (379 non zero parameters).

```python
def build_single_digit_mod_transformer():
    params = base_params(vocab_size=10, block_size=2, d_model=4)

    tok_emb = np.empty((10, 4))
    for i in range(10):
        x = i / 100
        a = (-x - 1 + (-3 * x ** 2 - 2 * x + 5) ** 0.5) / 2
        tok_emb[i] = np.array([x, 1, a, -x - a - 1])
    params['tok_emb'] = params['out_emb'] = tok_emb

    layer0 = base_layer(n_heads=1, d_model=4, d_ff=3 * 19, d_head=4)
    layer0['V'][0, :2, :2] = np.eye(2) * 1e4  # * 1e4 for messy cleanup.
    layer0['V'][0, 0, 0] *= 2 * 100  # * 2 for averaging, * 100 for tok_emb factor.
    layer0['P'][0, :2, :2] = np.eye(2)

    x0s = [layer_norm(np.array([i * 1e4, 1e4, 0., 0.]))[0] for i in range(19)]
    a0s = [tok_emb[i % 10] * 1e10 for i in range(19)]
    layer0['M1'][0], layer0['b1'], layer0['M2'], layer0['b2'] = build_MLP(x0s, a0s)

    params['layers'].append(layer0)

    return params
```

Figure 6.2: A Transformer for single digit addition mod 10.

6.7 Multi-Digit Modular Addition

Let's now turn our attention (pun not intended) to multi-digit addition (still mod 10). Here are example inputs and outputs for 2 digit numbers:

Input: `[1, 2, '+', 3, 3, '=']`

Output: `5`

Note that the output here is still just *a single* digit[‡]. We'll use a simple tokenizer over a vocabulary of size 12, where tokens 0-9 will correspond to the digits they represent, and '+' and '=' will be represented by 10 and 11 respectfully.

We'll be able to mostly reuse our construction for single digit addition, with some important modifications. First, we'd need to include position encoding and therefore increase the model size to 7.

Next, we'll now use proper cleanup for the attention module, otherwise, without proper care, we might run into precision issues for the MLP module (see Exercise 13). Note that since $d_{model} = 7$ in this case, but $d_{head} = 4$, and since we aren't attending to ourselves otherwise, if we actually wanted to cleanup everything we'd have no choice but "waste" 2 attention heads just for cleanup. Instead, since we'd like to keep the value and fixed constant 1, and since the position encoding are already normalized (so we can keep them without affecting normalization) it's enough to only clean up the two padding dimensions for the token embedding (a and $-x - a - 1$ above), so with proper care a single cleanup head suffices. With proper cleanup we can drop the large factors everywhere, which is nice.

Putting everything together, Figure 6.3 performs multi-digit additions mod 10.

This time, for adding single digit numbers, we require 7 model dimensions, 1,226 parameters (432 of these are non-0). For 2 digit numbers, still with 7 model dimensions, we'd need 9,508 parameters (3,646 are non zero, and almost all of them, 3,522, are part of the MLP module). Scaling to adding 3 digit numbers, mod 10, requires 90,690 parameters (or 35,542 non zero parameters), and again almost all of them (89,962 or 35,382 for the non zero ones) are used for the MLP module. Note that the minimum separation between input values for the MLP module (the `min_sep` variable in the `build_MLP` function) becomes very small (around 1.4e-9) for 3 digit additions (see Exercise 15).

‡ The output technically depends only on the least significant digits of the input numbers, so we could just ignore the other digits and solve it exactly as above. As we are building towards the general addition construction we will not do that (if it really bothers you, imagine the addition is performed mod 7 instead of mod 10).

```python
from itertools import product

def build_multi_digit_mod_transformer(n_digits):
    vocab_size = 10 + 2  # 10 digits, '+' (10) and '=' (11).
    block_size = 2 * n_digits + 2  # n_digits for each number +2 for '+' and '='.
    d_model = 7  # 4 for token, 3 for position.
    params = base_params(vocab_size, block_size, d_model)

    tok_emb = np.empty((vocab_size, 4))
    for i in range(10):
        x = i / 100
        a = (-x - 1 + (-3 * x ** 2 - 2 * x + 5) ** 0.5) / 2
        tok_emb[i] = np.array([x, 1, a, -x - a - 1])
    tok_emb[10] = tok_emb[11] = tok_emb[0]  # Anything normalized.
    tok_emb = np.pad(tok_emb, ((0, 0), (0, 3)))
    params['tok_emb'][:] = params['out_emb'][:] = tok_emb
    params['pos_emb'][:] = np.pad(pos_enc_3d_array(2 * n_digits + 2), ((0, 0), (4, 0)))

    out_range = 2 * (10 ** n_digits) - 1
    n_heads = 2 * n_digits + 1  # 1 cleanup head.
    layer0 = base_layer(n_heads, d_model, d_ff=3 * out_range, d_head=3)

    for head, (unit_pos, digit) in enumerate(product([1, n_digits + 2], range(n_digits))):
        theta = -(unit_pos + digit) * 2 * np.pi / block_size
        layer0['Q'][head, -3:, -3:] = pos_3d_enc_rotation_matrix(theta) * 1e6
        layer0['K'][head, -3:, -3:] = np.eye(3)
        layer0['V'][head, 0, 0] = 100 * (10 ** digit)
        layer0['P'][head, 0, 0] = 1

    # Cleanup head:
    layer0['Q'][-1, -3:, -3:] = (pos_3d_enc_rotation_matrix(0) * 1e6)
    layer0['K'][-1, -3:, -3:] = np.eye(3)
    layer0['V'][-1, 2:4, :2] = -np.eye(2)
    layer0['P'][-1, 2:4, :2] = np.eye(2)

    x0 = lambda i: np.concatenate((np.array([i, 1., 0, 0]), params['pos_emb'][0, -3:]))
    x0s = [layer_norm(x0(i))[0] for i in range(out_range)]
    a0s = [tok_emb[i % 10] * 1e10 for i in range(out_range)]
    layer0['M1'][0], layer0['b1'], layer0['M2'], layer0['b2'] = build_MLP(x0s, a0s)

    params['layers'].append(layer0)

    return params
```

Figure 6.3: A Transformer for multi-digit addition mod 10.

6.8 Full Construction

First, note that if we wanted to output just the *tens* digit, or just the *hundreds* digit, we'd need to change a single line in Figure 6.3 (Exercise 14).

Now, the first thing we'd need to address for the full DECIMAL ADDITION transformer is the way we read values with our attention mechanism attending to relative positions. Specifically, note that the relative distance between the output digit and the input digits changes after each digit we output (it increases by 1). As a simple solution in our case, we'll read values from absolute positions instead of from relative positions (see Exercise 16).

Since our MLP can translate any one dimensional value to any output, our strategy here would be to simply encode both the numerical value as well as the output digit position into this single value. The one thing to ensure is that the minimum separation between encoded values is not too small (see Exercise 17). To encode the position into this single dimension, we'll just sum the first dimension of the position embedding to our result number (see Exercise 18).

And... That's it folks! Figure 6.4 puts everything together. I have tested this on numbers of up to 3 digits each (so the result is up to 4 digits). How small are our programs? Well, for adding 1-digit numbers this construction needs 2,088 parameters (737 of these are non zero), for adding 2-digit numbers we need 27,432 parameters (10,440 are non zero), and finally, for adding up 3-digit numbers we need 360,576 parameters (139,267 are non zero). Note that almost the entirety of the parameters in this construction, especially for the larger versions, lie in the MLPs:

Inputs	$\lvert\{param\}\rvert$	$\lvert\{param \neq 0\}\rvert$	$\lvert\{param \in MLP\}\rvert$	$\lvert\{param \neq 0 \in MLP\}\rvert$
1-digit numbers	2,088	737	1,717 (82.2%)	648 (87.9%)
2-digit numbers	27,432	10,440	26,872 (98.0%)	10,326 (98.9%)
3-digit numbers	360,576	139,267	359,827 (99.8%)	139,128 (99.9%)

Note that in all cases we are using just 7 model dimensions, but the very high dimensional MLPs (much more than the standard $4 \cdot d_{model}$) would force our parameter count to go way up had we used the standard configuration.

```python
def build_decimal_addition_transformer(n_digits):
    vocab_size = 10 + 2  # 10 digits, '+' (10) and '=' (11).
    block_size = 2 * n_digits + 2 + n_digits  # An extra n_digits for the result.
    d_model = 7  # 4 for token, 3 for position.
    params = base_params(vocab_size, block_size, d_model)

    tok_emb = np.empty((vocab_size, 4))
    for i in range(10):
        x = i / 100
        a = (-x - 1 + (-3 * x ** 2 - 2 * x + 5) ** 0.5) / 2
        tok_emb[i] = np.array([x, 1, a, -x - a - 1])
    tok_emb[10] = tok_emb[11] = tok_emb[0]  # Anything normalized.
    tok_emb = np.pad(tok_emb, ((0, 0), (0, 3)))
    params['tok_emb'][:] = params['out_emb'][:] = tok_emb
    params['pos_emb'][:] = np.pad(pos_enc_3d_array(3 * n_digits + 2), ((0, 0), (4, 0)))

    out_range = 2 * (10 ** n_digits) - 1
    n_heads = 2 * n_digits + 1  # 1 cleanup head.
    layer0 = base_layer(n_heads, d_model, d_ff=(n_digits + 1) * 3 * out_range, d_head=3)

    for head, (num_pos, digit) in enumerate(product([0, n_digits + 1], range(n_digits))):
        layer0['Q'][head, 1] = params['pos_emb'][num_pos + n_digits - 1 - digit, -3:] * 1e6
        layer0['K'][head, -3:, -3:] = np.eye(3)
        layer0['V'][head, 0, 0] = (10 ** digit) * 100
        layer0['P'][head, 0, 0] = 1 / 100

    # Cleanup and mix head:
    layer0['Q'][-1, -3:, -3:] = (pos_3d_enc_rotation_matrix(0) * 1e6)
    layer0['K'][-1, -3:, -3:] = np.eye(3)
    layer0['V'][-1, 2:4, :2] = -np.eye(2)
    layer0['P'][-1, 2:4, :2] = np.eye(2)
    layer0['V'][-1, 0, -1] = -1
    layer0['V'][-1, -3, -1] = 1
    layer0['P'][-1, 0, -1] = 1

    x0 = lambda i, p: np.concatenate((np.array([i / 100 + p[-3], 1., 0, 0]), p))
    x0s, a0s = [], []
    for i in range(out_range):
        for p in params['pos_emb'][-n_digits-1:, -3:]:
            x0s.append(layer_norm(x0(i, p))[0])
        for digit in range(n_digits, -1, -1):
            a0s.append(tok_emb[i // (10 ** digit) % 10] * 1e10)

    layer0['M1'][0], layer0['b1'], layer0['M2'], layer0['b2'] = build_MLP(x0s, a0s)

    params['layers'].append(layer0)

    return params
```

Figure 6.4: A Transformer for Decimal Addition.

6.9 Exercises

♙ 1. Show that given two sets of n and m vectors in $\mathbb{R}^d$, $v_1, \ldots, v_n$ and $u_1, \ldots, u_m$, the space spanned by the $n \cdot m$ vectors $\{v_i + u_j | , 1 \leq i \leq n, 1 \leq j \leq m\}$ is at most $n + m - 1$ dimensional.

♙ 2. Prove Corollary 6.1.

♙ 3. Prove Theorem 6.1.

♙ 4. Consider the following definition:

Definition 6.2 (Weak Spike)**.** *A function $f : \mathbb{R} \to \mathbb{R}$ is a **weak spike** if $f(x) \leq 0$ if $x \leq -1$ or $x \geq 1$, and $f(x) = 1$ if $x = 0$.*

Show that you can improve the construction from Lemma 6.1 to only use 2 hidden MLP dimensions if you only need to construct a weak spike instead of a standard spike.

♙ 5. How many parameters would our construction for single digit modular addition (Figure 6.2) require if we used base b instead of base 10?

♙ 6. Implement the construction from Section 6.1 outputting $(1 - k, k, 0, 0, \ldots)$.

7. Re-implement the construction for single figure addition from Figure 6.2 using just 3 model dimensions. What happened to the total parameter count?

8. Re-implement the construction for single figure addition from Figure 6.2 using *proper cleanup* (Section 3.4).

9. Show that the matrix $\begin{bmatrix} \delta_0 & \delta_1 \cdots \cdots \delta_{n-1} & \delta_n \\ \delta_1 & \delta_0 \cdots \cdots \delta_{n-2} & \delta_{n-1} \\ \vdots & \vdots & \vdots \\ \delta_{n-1} & \delta_{n-2} \cdots \cdots \delta_2 & \delta_1 \\ \delta_n & \delta_{n-1} \cdots \cdots \delta_1 & \delta_0 \end{bmatrix}$ where the (i, j)th element equals $\delta_{|i-j|} = cos(\alpha|i - j|)$ for some α, is of rank at most 2 (hint: you can show something much more general).

♙ 10. Make the economic construction from the end of Section 6.2 work by changing the position embeddings (hint: what happens if the position embeddings are not unique?).

♛ 11. Make the economic construction from the end of Section 6.2 work *without changing the position embeddings* - i.e. use only one attention head for reading 2 digits in the same decimal position from both input numbers in parallel for *multi digit numbers* (note that Section 6.2 and Figure 6.2 emplyed this trick only for single digit numbers). How many more model dimensions did you need?

𝒴 12. Does the proof from Section 6.1 hold when you take into account the residual connection?

13. Get the construction for the multi-digit addition mod 10 (Figure 6.3) to work without the extra proper cleanup head in the attention layer. How many more parameters do you

need?

♙ 14. Change one line in Figure 6.3 so that the output is just the tens-digit of the addition, i.e. if we're adding a and b, output $\lfloor \frac{a+b}{10} \rfloor \pmod{10}$.

15. Change the code in Figure 6.3 such that `min_sep` is at least a couple of orders of magnitude higher for 3 digit numbers. What is it for 5 digit numbers? How many extra parameters did you need?

♛ 16. Change our solution for Decimal Addition (Figure 6.4) to allow for an arbitrary prefix, that is ignored, before the input. Specifically, attend to relative positions instead of absolute positions.

17. Add multiplicative constants to both value and position parameter in Figure 6.4 to maximize `min_sep`.

18. What happens to our construction in Figure 6.4 if we use the first dimension of the position embedding instead of the third? Is there a better linear combination?

𝒴 19. Change our construction of Decimal Addition to use no zero padding. You can still expect your input digits to be up to a certain pre-specified length, but they can be shorter. Likewise, don't output padding zeros in your result. Remember to increase the vocabulary size to 14 (add 2 new symbols for the sentinel tokens `<bos>` and `<eos>`). Your Transformer should support both `4+8=12` and `84+88=172`.

𝒴 20. Implement Decimal Multiplication which performs, well, decimal multiplication instead of addition.

Part III

Further Topics

Chapter 7

The OG Encodings

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

BRIAN KERNIGHAN

So far in this booklet we have not made use of the original sinusoidal embeddings* from the original Transformer paper [Vaswani et al., 2017] (henceforth we'll refer to them simply as *the OG embeddings*). Why is that? We did try to make our programs efficient, and using the OG sinusoidal embeddings instead of "learned" embeddings would have saved us all of the position embedding parameters - for our toy programs, often a considerable part! We will dedicate this section to considering the OG embeddings more deeply and to then answering this question.

Let's start by recalling the definition of the OG encodings:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

* As you may have noticed by now, I am using the words "encodings" and "embeddings" interchangeably throughout this work.

Figure 7.1 shows a simple implementation for the OG embedding, and Figure 7.2 shows an example of the OG encoding for 64 positions and 128 dimensions.

```
def og_embedding(block_size, d):
    t = np.arange(block_size)[:, None] / (10_000 ** (2 * np.arange(d // 2) / d))[None, :]
    res = np.empty((block_size, d))
    res[:, 0::2] = np.sin(t)
    res[:, 1::2] = np.cos(t)
    return res
```

Figure 7.1: The OG embedding.

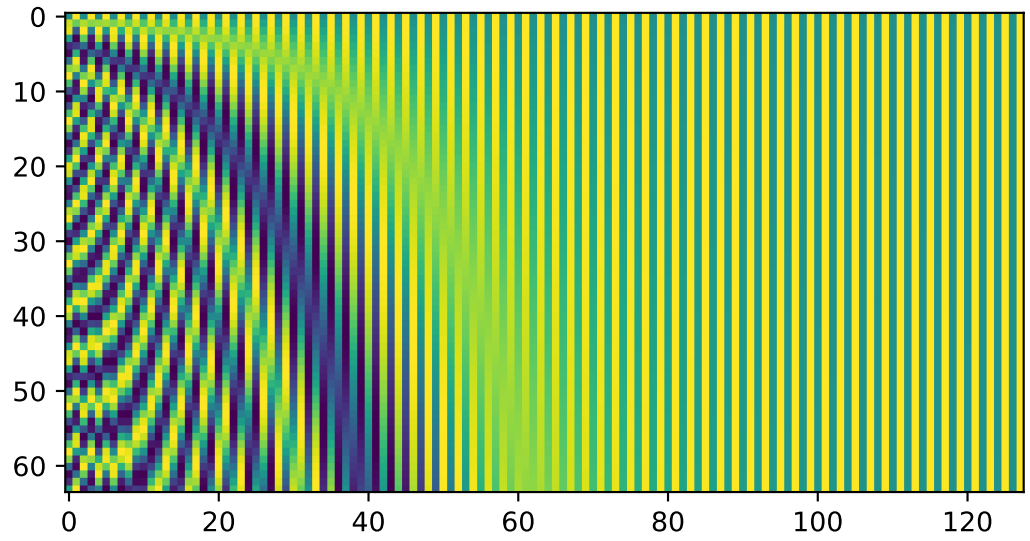


Figure 7.2: The OG embedding with a block size of 64 and d_{model} 128.

Recall that we discussed three properties of embeddings: (1) being fixed under layer normalization (Section 2.3), (2) being *linearly shiftable* which allows easy access to relative positions (Section 3.2), and (3) leaving enough free space for computation (Section 4.1). We'll soon consider the OG embeddings from these perspectives, and then we'll define and investigate a fourth interesting property.

First though, we'll define a generalization and reformulation of the OG embedding that will make the rest of discussion much easier.

Beware - spoilers ahead! If you haven't yet and would like to think a bit more about the OG embeddings (Exercise 1) now would be a good time.

7.1 Rotation Encoding

Let's start with a couple of basic definitions:

Definition 7.1 (Real Encoding)**.** *A **real encoding of size q into d dimensions** is simply a function $f : \mathbb{Z}_q \to \mathbb{R}^d$.*

Definition 7.2 (Complex Encoding)**.** *A **complex encoding of size q into d dimensions** is simply a function $g : \mathbb{Z}_q \to \mathbb{C}^d$.*

As in Chapter 3 I'm using the notation $\mathbb{Z}_q$ as a shorthand for the set $\{0, \ldots, q-1\}$ denoting our vocabulary. Given a vocabulary V of q elements, $v_1, \ldots, v_q$ and an embedding f of size q (real or complex) we'll colloquially talk about applying the embedding to V by sending v_i to $f(i-1)$.

Note that every complex encoding into $\frac{d}{2}$ dimensions *induces* a real encoding into d dimensions, with the natural mapping of a complex number into its real and imaginary parts, and vice versa. We'll usually prefer the standard mapping $z \to (\Re(z), \Im(z))$ where the real part comes first, except when we'll deal with the OG positional encoding, where having the imaginary part before the real part will make more sense. For a complex encoding $g : \mathbb{Z}_q \to \mathbb{C}^d$ we'll use the notation $g^{\mathbb{R}} : \mathbb{Z}_q \to \mathbb{R}^{2d}$ to denote its induced natural real counterpart. In code, we'll represent encodings as 2-dimensional arrays of shape `(q, d)` with the appropriate (real or complex) types.

We're now ready for the main definition of this section:

Definition 7.3 (Rotation Encoding)**.** *Let $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_n) \in [0, 2\pi)^n$ be a vector of n angles. Define the **complex rotation encoding** $RE_\theta : \mathbb{Z}_q \to \mathbb{C}^n$ as the complex encoding such that $RE_\theta(a) = (e^{ia\theta_1}, \ldots, e^{ia\theta_n})$. Given a complex rotation encoding RE_θ, the induced **rotation encoding** is the induced real rotation encoding $RE_\theta^{\mathbb{R}}$.*

Note that, like many of the terms in this booklet "rotation encoding" is not part of the standard terminology (don't confuse it with the similarly named Rotary Position Embedding [Su et al., 2022]!).

```python
def rotation_encoding(q, thetas):
  return np.exp(np.arange(q)[:, None] * 1j * thetas[None, :]).view(dtype=np.float_)

def reversed_rotation_encoding(q, thetas):
  vs = np.exp(np.arange(q)[:, None] * 1j * thetas[None, :])
  res = np.empty((q, thetas.shape[0] * 2))
  res[:, 0::2] = vs.imag
  res[:, 1::2] = vs.real
  return res
```

Figure 7.3: An implementation of rotation encoding and reverse rotation encoding in Numpy.

See Figure 7.3 for an implementation generating the induced real rotation encoding given an angles vector `thetas`. It also includes a slightly lengthier implementation of reversed rotation encoding - which are the induced real encoding in case you put the imaginary part before the real part. Note that all of the discussion in this chapter holds for both types of embeddings and we'll usually just say rotation embedding when we refer to both.

An equivalent view of a rotation encoding is to take a vector whose coordinates are n complex units $\mathbf{v} = (e^{i\theta_1}, \ldots, e^{i\theta_n}) \in \mathbb{C}^n$, and define $RE_{\mathbf{v}}(a) = \mathbf{v}^a$, where the exponentiation is done element-wise.

For example, for any complex rotation encoding $RE_{\mathbf{v}}$, we have $RE_{\mathbf{v}}(1) = \mathbf{v}$ and $RE_{\mathbf{v}}(0) = (1, \ldots, 1) \in \mathbb{C}^n$ (and for the induced real rotation encoding, $RE^{\mathbb{R}}_{\mathbf{v}}(0) = (1, 0, \ldots, 1, 0) \in \mathbb{R}^{2n}$). Since the elements of a complex rotation encoding are complex units, we get the following:

Theorem 7.1 (Norm of Rotation Encoding)**.** *For any $a \in \mathbb{Z}_q$ and complex rotation encoding $RE : \mathbb{Z}_q \to \mathbb{C}^d$, we have $|RE(a)|^2 = d$. For any $a \in \mathbb{Z}_q$ and induced real rotation encoding $RE^{\mathbb{R}} : \mathbb{Z}_q \to \mathbb{R}^d$, we have $|RE^{\mathbb{R}}(a)|^2 = \frac{d}{2}$.*

In Section 3.2 we defined *linearly shiftable encodings*. We can easily extend the definition to complex encodings.

Definition 7.4 (Linearly Shiftable Complex Encodings)**.** *A complex encoding $f : \mathbb{Z}_q \to \mathbb{C}^d$ is **linearly shiftable** if for every $\delta \in \mathbb{Z}$ there exists a linear transformation over $\mathbb{C}$, $T_\delta : \mathbb{C}^d \to \mathbb{C}^d$, such that for all $j \in \mathbb{Z}_q$ and $0 \leq j + \delta < q$ we have $T_\delta f(j) = f(j + \delta)$. We'll call T_δ the **shifting transformation** of f by δ.*

It immediately follows that:

Lemma 7.1. *If a complex encoding is linearly shiftable then its corresponding real encoding is linearly shiftable.*

Proof. Exercise 2. □

What's so special about rotation encodings? Well, first, note the following useful property:

Theorem 7.2 (Rotation Encodings are Linearly Shiftable)**.** *Let $RE_\theta : \mathbb{Z}_q \to \mathbb{C}^d$ be a complex rotation encoding. Then RE_θ is linearly shiftable.*

Proof. Given $\delta \in \mathbb{Z}$, let T_δ be the linear transformation defined by the diagonal matrix $M_\delta = \begin{pmatrix} e^{i\delta\theta_1} & & \\ & \ddots & \\ & & e^{i\delta\theta_d} \end{pmatrix}$. Then T_δ linearly shifts RE_θ by δ. □

From Theorem 7.2 and Lemma 7.1, real encodings induced by rotation encodings are also linearly shiftable. See Figure 7.4 for an implementation of the shifting construction for the induced real rotation encoding from Theorem 7.2 and Lemma 7.1.

```
def shift_by_delta(d, delta, theta):
  M = np.zeros((d, d))
  for i in range(d // 2):
    M[i * 2,     i * 2]     =  np.cos(theta[i] * delta)
    M[i * 2 + 1, i * 2]     = -np.sin(theta[i] * delta)
    M[i * 2,     i * 2 + 1] =  np.sin(theta[i] * delta)
    M[i * 2 + 1, i * 2 + 1] =  np.cos(theta[i] * delta)
  return M
```

Figure 7.4: An implementation creating the shifting matrix for $\delta =$ `delta` for the induced real encoding in dimension `d` from a rotation encoding with angles `theta` in Numpy.

From now on we'll only deal with real encodings.

7.2 Back to the OG Encoding

Ok - so now for the real reason we care about rotation encodings:

Theorem 7.3 (OG Encodings are Reversed Rotation Encodings)**.** *The positional encoding suggested in the original Transformer paper (the* OG encodings*) in dimension $2d$ is the induced reversed real rotation encoding from a complex rotation encoding with angles $\boldsymbol{\theta_{OG}} = (\frac{1}{10{,}000^{\frac{0}{d}}}, \frac{1}{10{,}000^{\frac{1}{d}}}, \ldots, \frac{1}{10{,}000^{\frac{d-1}{d}}})$.*

Proof. Exercise 4. Note that in the original Transformer paper, the authors map a complex number c to $(\Im(c), \Re(c))$, i.e. the order of the real and imaginary parts are reversed. □

We can verify:

```
og_angles = lambda d: 1 / (10_000 ** (2 * np.arange(d // 2) / d))

q = 64
d = 128
np.testing.assert_allclose(reversed_rotation_encoding(q, og_angles(d)), og_embedding(q, d))
```

Figure 7.5: The OG encodings are reversed rotation encodings.

Putting the above together, we immediately get the following fact stated in the Transformer paper:

Corollary 7.1. *The OG encoding is linearly shiftable.*

Great, so we just proved the claim from the original Transformer paper [Vaswani et al., 2017] that the OG embeddings are linearly shiftable[†]. In fact, we've shown something more general.

Alright, so *why* then, did we not use the OG encoding then?

One reason, is that the OG encoding needs a minimum of 4 dimensions (as the number of dimensions must be even, and we already know that 2 dimensions are not enough, Section 2.3).

A second reason is that some constructions (e.g. Hello World) are much easier with our pseudo "learned" embeddings.

But the most important reason is different. The OG embeddings are *not* fixed under layer normalization. Worse, once layer normalization is applied, the OG embedding are no longer linearly shiftable (Exercise 5). Wait, so how did they work for attending to relative positions in the original Transformer paper?

The reason they worked in the original Transformer paper, is that there, the authors used PostLN - so the attention module saw the original *non-normalized* (and therefore still linearly shiftable) embeddings. Note that according to Theorem 7.1, the length of all OG embeddings of dimension d *is* constant ($\frac{d}{2}$) so if we used RMSNorm [Zhang and Sennrich, 2019] for

[†] Note that the term "linear shiftability" is not standard, and they don't use it - I made it up for this book.

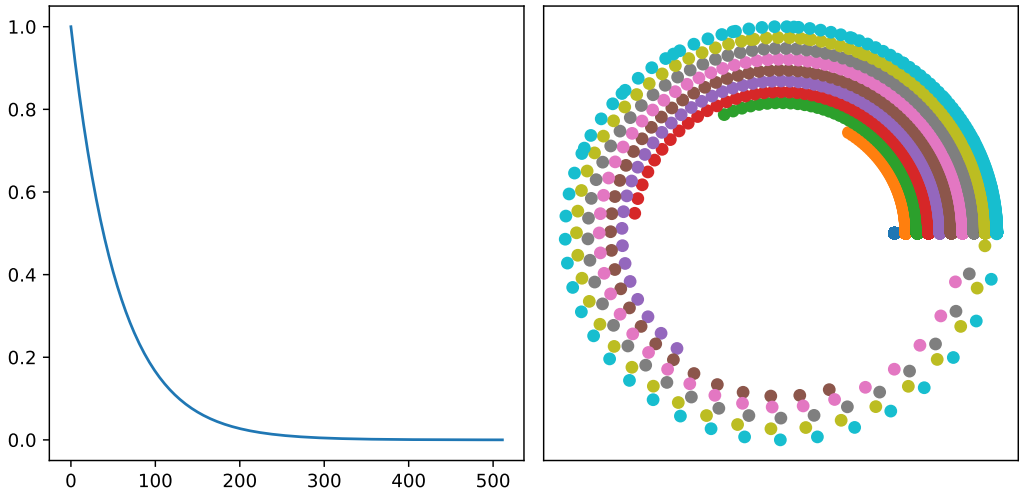


Figure 7.6: The OG angles for $d_{model} = 1,024$, i.e. 512 angles (Left). The first 10 positions, depicted as complex numbers (Right).

example, all embedding vectors would be scaled by a constant factor (Exercise 6). What would that mean about attending to relative positions with the OG embeddings and PostLN, but using RMSNorm instead of LayerNorm? (See Exercise 6).

Rotation encodings, taking basically a single line of code to implement (see Figure 7.3), seem natural and nice. One thing still seems very arbitrary though about the OG embeddings - the angles!

Figure 7.6 shows the OG angles visually. What do you think about it? We'll revisit this figure soon.

What could we use instead of the OG angles? We could for example just sample the angles uniformly at random from $[0, 2\pi]$. Let's call such rotation embeddings *random rotation encodings*.

Definition 7.5 (Random Rotation Encoding)**.** *A* random rotation encoding *of dimension* $d \in \mathbb{Z}^+$ *is a rotation encoding where the angles are sampled independently uniformly at random from* $[0, 2\pi]$*, i.e.* $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_{\frac{d}{2}})$ *amd* $\theta_i \sim U(0, 2\pi)$*.*

Figure 7.7 shows an implementation, and Figure 7.8 shows the actual embeddings for 64 positions and 128 dimensions.

Note that from Theorem 7.2 we get that the random rotation encoding are linearly shiftable as well.

```python
def random_rotation_embedding(q, d):
    return rotation_encoding(q, thetas=np.random.uniform(0, 2 * np.pi, size=d // 2))
```

Figure 7.7: Random rotation embeddings.

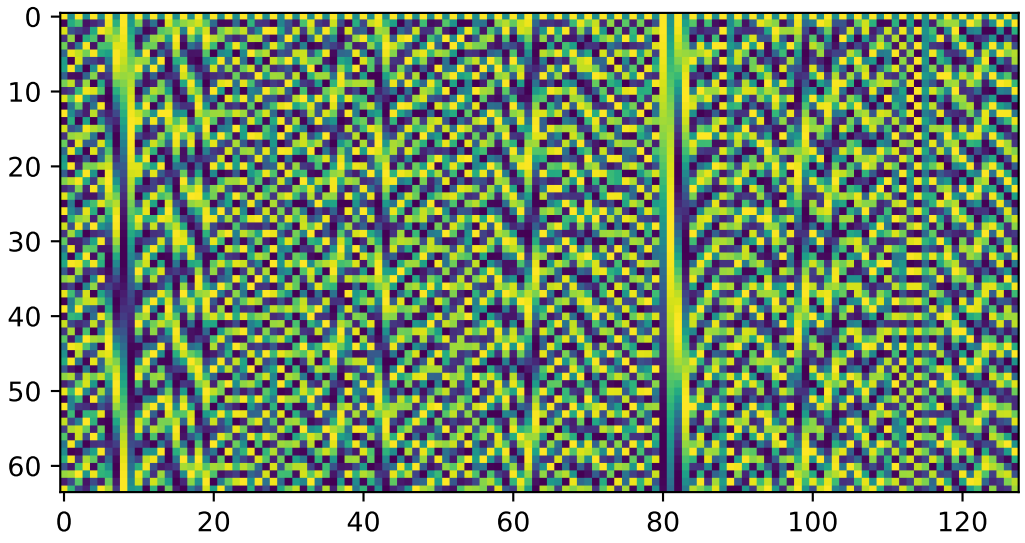


Figure 7.8: Random rotation embedding for 64 positions and 128 dimensions (i.e. 64 angles).

What about using evenly spaced angles instead of random ones? Wouldn't that be even better? In Exercise 8 you'll show that would be a bad idea.

So is there any reason to prefer one set of angles over other? We'll answer this question in the next two sections.

7.2.1 Distinguishability

We'll now turn our attention to a new property of encodings - *distinguishability*, which will allow us to attend and read information with a low amount of noise, complementing attention hardening (Section 3.1).

Definition 7.6 (Distinguishability)**.** *The* ***distinguishability*** D *of an encoding* $f : \mathbb{Z}_q \to \mathbb{R}^d$ *satisfies* $D = \min(diag(softmax(F)))$*, where* $F_{i,j} = f(i) \cdot f(j)$ *and diag returns the elements on the diagonal of a matrix.*

Take a position encoding $f : \mathbb{Z}_q \to \mathbb{R}^d$ with distinguishability D. Now consider trying to read a value from a specific previous position, without attention hardening (Section 3.1). Distinguishability provides some way to measure the amount of noise that we'll get. Specifically, if both our query and key are exactly equal to the encoding of the minimum element on the softmax matrix diagonal from Definition 7.6 (i.e. without the large factor of attention hardening), the value that we'll end up getting will be $D \cdot v_{correct} \times (1 - D)v_{noise}$. Can we artificially increase distinguishability just by scaling the position embedding? Yes with PostLN, but not with PreLN. By applying attention hardening like we have throughout this book we can emulate an increased distinguishability, but note that that requires extremely high weights. So high distinguishability could be desired in order to harden attention without using large factors from the attention hardening trick we developed in Section 3.1.

Note that while we presented high distinguishability as a desired trait, it might sometimes be helpful to have low distinguishability between related elements. For example, for fuzzy tasks like understanding language, when trying to read a position k places back, some "noise" from its neighboring positions might be helpful.

For now, consider Figure 7.9 which compares the distinguishability of the OG encoding and a random rotation encoding for 256 elements, with a varying number of dimensions. We see that the random encoding has much higher distinguishability (> 0.99 starting at around 10 dimensions) whereas OG encoding only has distinguishabilty of 0.94 with 256 dimensions.

Indeed, note that since many of the angles θ_i from the OG encodings are small (close to 0), the resulting encoding results in a very soft attention (low distinguishability). For example,

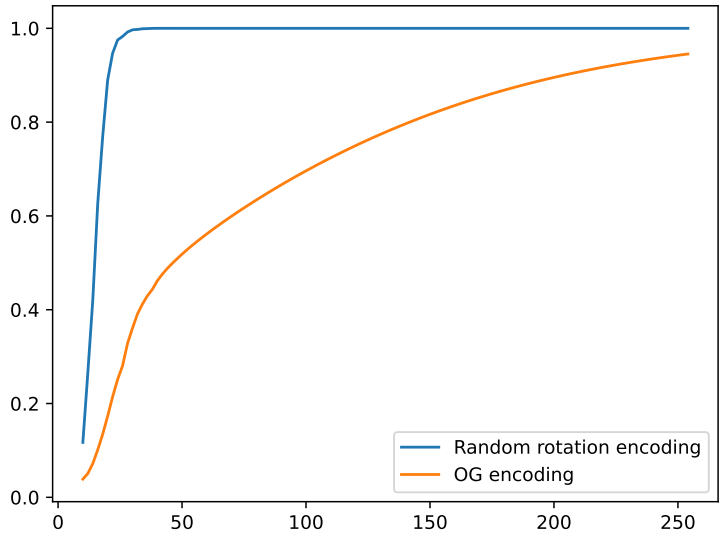


Figure 7.9: A comparison of the distinguishability of the OG encoding and a random encoding for 256 positions as a function of the number of dimensions.

define:

$$M = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$$

Then with just 64 positions and $d_{model} = 128$ (double than needed even for sparse encoding), we have $softmax(MM^T)_{ii} \approx 0.77 \ll 1$.

And as we saw, we could substantially increase the hardness of the attention if we replaced the OG angles θ_i above with e.g. ones selected at random $\hat{\theta}_i \sim U(0, 2\pi)$. That said, with the attention hardening trick (see 3.1) the original encoding and angles will suffice. Specifically, with the original angles and attention hardening, a block size of 256, and 2 dimensions of original position embedding (and without layer normalization), we can get the equivalent of hard positional attention for all practical purposes, with a large enough N value of say 1,000,000.

The distinguishability of a random embedding is much higher than that of the OG embedding. For now 1-0 for the random embedding. Do the OG embedding have anything useful going on for them?

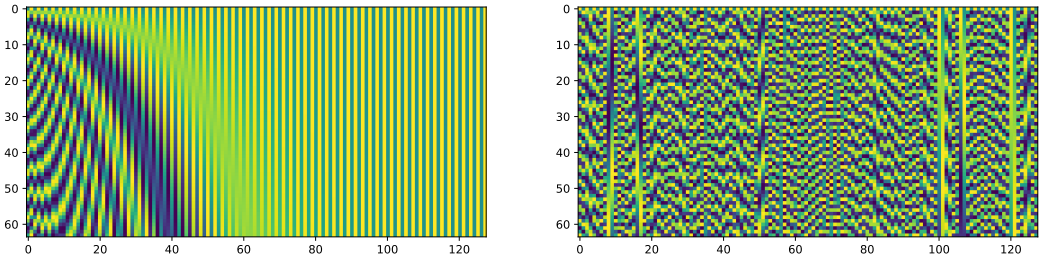


Figure 7.10: The OG embedding (Left) and a random rotation embedding (Right) for 64 positions and 128 dimensions.

7.2.2 Free Space

For easier reference, Figure 7.10 compares the OG embeddings and a random rotation embeddings side by side. Note that the right side (higher dimensions) of the OG embeddings seems almost *independent of the encoded position* - a useful property allowing for "free space" for processing. In constrast, the random encoding in the right side of Figure 7.10 seems to have very few columns that seem (almost) independent of the encoded element. Could this be a redeeming quality of the OG embedding?

Let's make more formal the intuition that the OG encoding has a bunch of "free space" whereas a random rotation encoding does not.

Definition 7.7 (Padding Dimension)**.** *An encoding $f : \mathbb{Z}_q \to \mathbb{R}^d$ has **padding of dimension at least k with ϵ noise** if there is a subspace $K \subseteq \mathbb{R}^d$ of dimension k and a vector $v \in R^k$ such that for all $a \in \mathbb{Z}_q$ we have $|f(a)_K - v| \leq \epsilon$ where $f(a)_K$ denotes the projection of $f(a)$ onto K. If K is parallel to the standard axes, we'll say that the padding is **standard**. If $v = 0$ we'll say that the padding is **simple**. If an encoding has padding of dimension at least k but not padding of dimension at least $k + 1$ we'll say that the encoding has **padding dimension k**. If an encoding has a standard padding of dimension at least k but not a standard padding of dimension at least $k + 1$ we'll say that the encoding has **standard padding dimension k**.*

With this definition, we immediately get the following lemma:

Lemma 7.2. *The standard padding dimension with noise ϵ of an encoding $f : \mathbb{Z}_q \to \mathbb{R}^d$ equals $|\{0 \leq i < d \mid max(f(\mathbb{Z}_q)_i) - min(f(\mathbb{Z}_q)_i) \leq 2\epsilon\}|$.*

Using Lemma 7.2 we can calculate the standard padding dimension of the encodings we've encountered thus far. For example, with $\epsilon = 1e^{-3}$ and encoding $\mathbb{Z}^{64}$ into $\mathbb{R}^{128}$, the OG encodings have a standard padding dimension of 16, while a random rotation encoding only

```python
def standard_padding_dimension(encoding, epsilon=1e-3):
  return np.sum(np.max(encoding, axis=0) - np.min(encoding, axis=0) <= 2 * epsilon)
```

Figure 7.11: Calculating the standard padding dimension of an encoding.

has a standard padding dimension of 0 (this can be verified empirically using the code from Figure 7.11).

Having a higher padding dimension means that we can write computation results or copy information from previous tokens without noise and without cleaning up (Section 3.4). Even more importantly, note that the first thing that the Transformer does is sum the position embeddings and the (learned) token embeddings - having a high padding dimension might make it easier to learn token embeddings in a way that is independent of the position. Throughout this book we've seen that cleanup was almost always free or at least very cheap. In real life situations though, where the attention heads often have much less dimensions than the model, cleaning up could be a more costly operation. So a high padding dimension could turn out useful (see Exercise 11).

7.3 Non Linearly Shiftable Encodings ϟ

Linear shiftability is a critical quality for enabling attending to relative positions (see Section 3.2). In this very optional section we will consider some more encodings which are not lineraly shiftable. As such, they might not be very helpful as positional encodings, but could be helpful to better understand how to embed tokens.

Definition 7.8 (Random Normal Encoding)**.** *Random normal encodings are encodings* $f : \mathbb{Z}_q \to \mathbb{R}^d$ *such that for each* $i \in \mathbb{Z}_q$*,* $f(i)_k \sim N(0, 1)$ *and the coordinates are i.i.d.*

Note that we already used random normal encodings, e.g. in the construction of the untied Transformer in Chapter 3 (Figure 3.3).

Definition 7.9 (Random Sign Encoding)**.** *Random sign encodings are encodings* $f : \mathbb{Z}_q \to \{-1, 1\}^d$ *such that for each* $i \in \mathbb{Z}_q$ *and* $0 \leq k < d$ *we have* $P(f(i)_k = 1) = \frac{1}{2}$ *and the coordinates are i.i.d.*

Figure 7.3 shows an implementation of random normal encoding and random sign encoding.

Finally we'll present Hamming code encoding:

Definition 7.10 (Hamming Code Encoding)**.** *Hamming code* (n, r) *encoding are encoding*

```python
def random_normal_encoding(q, d):
    return np.random.normal(0, 1, (q, d))

def random_sign_encoding(q, d):
  return np.random.randint(0, 2, (q, d)) * 2 - 1
```

Figure 7.12: An implementation of random normal encoding and random sign encoding in Numpy.

functions $f : \mathbb{Z}_q \to \{-1, 1\}^d$ *such that for each* $i \in \mathbb{Z}_q$ *and* $0 \leq k < d$ *we have* $f(i)_k = 1$ *iff.* $Hamming(n, r)_{i,k} = 1$.

Hamming encodings provide a guarantee for a certain level of distinguishability. Unfortunately, unlike random normal encodings or random sign encodings, they are highly unbalanced (i.e. their mean is far from 0). So also consider the (almost) balanced version, where the number of positive elements (+1s) is off by no more than 1 from the number of negative elements (-1s). Figure 7.14 show an implementation of Hamming encoding.

Non-linearly-shiftable encodings are less useful as positional encodings, as they don't easily support attending to relative positions, but they do support some beneficial properties making them potentially suitable to represent tokens.

7.4 Exercises

'...' 1. Consider the OG embeddings - can you see the motivations of the authors of the Transformer paper to have defined them as they did?

♙ 2. Prove Lemma 7.1.

♙♪ 3. Show that the OG embeddings are linearly shiftable.

♙ 4. Prove Theorem 7.3.

♙ 5. Show that the OG embeddings after layer normalization are *not* linearly shiftable.

6. What happens to the OG embeddings when applying RMSNorm [Zhang and Sennrich, 2019]? Can we easily use the OG embeddings to attend to relative positions with a PreLN Transformer if we use RMSNorm instead of LayerNorm?

♪ 7. Generalize the `shift_by_delta` function from Figure 7.4 to create a matrix that attends equally uniformly to k past position. I.e. if the keys are positional rotation encodings for positions $1, \ldots, i$, and the query is a position rotation encoding for position i, the attention weights should be 0 for positions $j \leq i - k$ and $\frac{1}{k}$ for any position $i - k < j \leq i$.

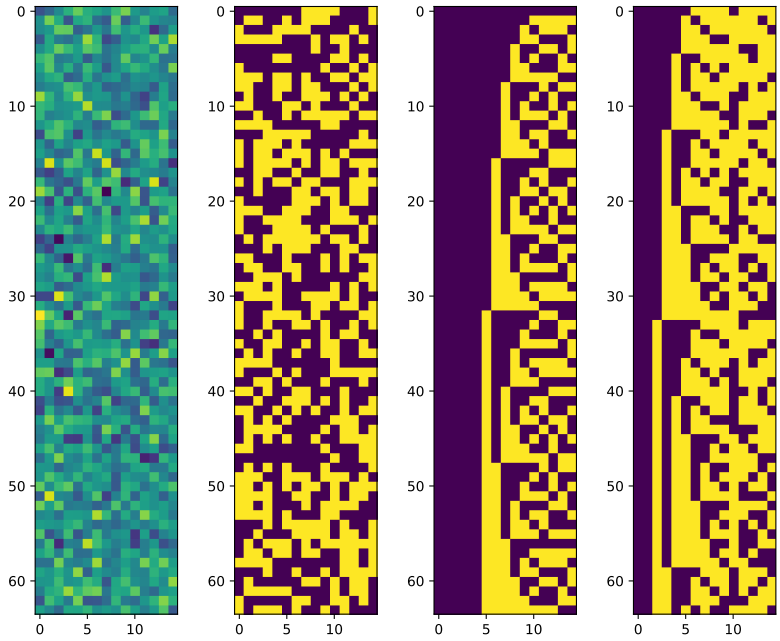


Figure 7.13: Examples of non linearly shiftable encodings: random normal encodings (left), random sign encoding (second from left) and Hamming encoding(15, 4) (second from right), and almost balanced Hamming encoding(15, 4) (right). All with 64 elements and 15 dimensions.

```python
import itertools

all_bit_strings = lambda n: list(itertools.product([0, 1], repeat=n))

def hamming_embedding(r):
    n = 2 ** r - 1
    k = n - r
    bits = [x for x in all_bit_strings(r) if x != (0,) * r]
    bits = sorted(reversed(bits), key=lambda s: s.count(1))  # Put in standard form.
    H = np.array(bits).T
    A = H[:, r:]
    G = np.concatenate((np.eye(k), A.T), axis=1)
    bits = np.array(all_bit_strings(k))
    return (np.einsum('kn,vk->vn', G, bits) % 2) * 2 - 1.
```

Figure 7.14: An implementation of Hamming encoding.

We'd like:

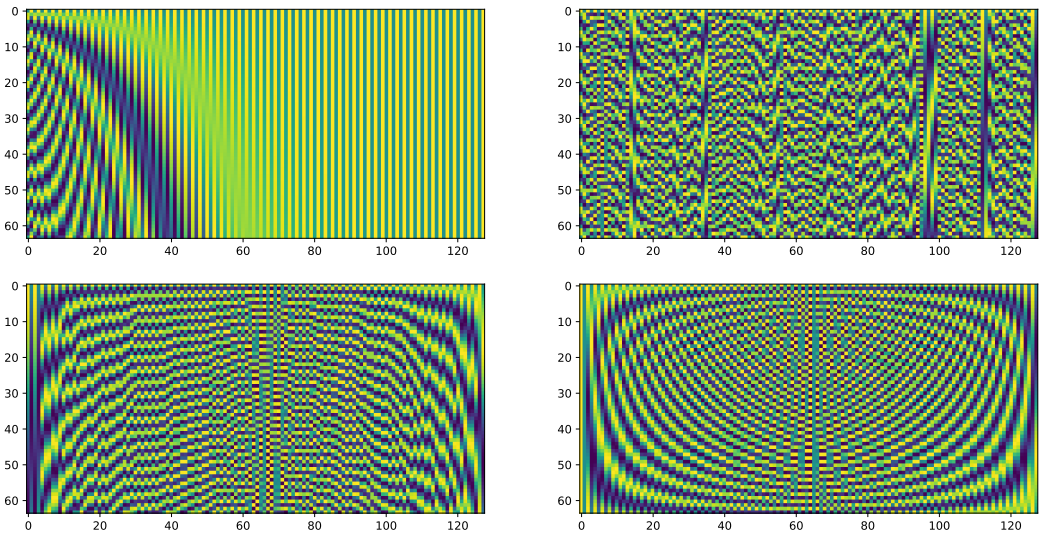
$$attention\_weight(i,j) = \begin{cases} \frac{1}{k} & i-k < j \leq i \\ 0 & j \leq i-k \end{cases}$$

Note that simply summing the matrices returned by `shift_by_delta` will not work.

8. What would be the distinguishability of a rotation encoding where the angles are evenly spaced, like in `uniform_rotation_embedding` below?

```
def uniform_rotation_embedding(q, d):
    return rotation_encoding(q, np.linspace(0, 2 * np.pi, d // 2, endpoint=False))
```

♙ 9. Here's a plot of four rotation embeddings: The OG embedding, a random rotation embedding, a random rotation embedding where the angles are sorted, and of uniform rotation embedding, all with 64 positions and 128 dimensions. Which is which? Explain the visual patterns.



♛ 10. Write a function to calculate the *padding dimension* of an embedding (as opposed to calculating the *standard* padding dimension in Figure 7.11).

11. Train a Transformer (with PostLN) on your favorite dataset, once with the OG embedding and once with random rotation embedding. Which one works best?

Chapter 8

The TAOTP Interpreter ϟ

To learn something, to master something, anything, is as sweet as first love.

GEOFFREY WOLFF

There is no great genius without some touch of madness.

ARISTOTLE

An expert is a man who has made all the mistakes which can be made, in a narrow field.

NIELS BOHR

THROUGHOUT this book, we set the weights for our Transformer programs via small Python programs - for example, see Figure 2.9. In this (truly skippable chapter) we will take things to the next mastery level and set the numerical weights directly manually.

```python
import ast

def compile_TAOTP(program: str):
    params = ast.literal_eval(program)
    if 'out_emb' not in params:
        params['out_emb'] = params['tok_emb']
    for emb in ['tok_emb', 'pos_emb', 'out_emb']:
        params[emb] = np.array(params[emb])
    for layer in params['layers']:
        for k, lst in layer.items():
            layer[k] = np.array(lst)
    return params
```

Figure 8.1: A simple interpreter for the TAOTP language.

Also, we mentioned that manually setting the weights of a Transformer could be seen as an esoteric programming language, so it would be nice to have an interpreter for that language (that can't just run any Python program!). You can find a simple implementation of such an interpreter for this language in Figure 8.1.

For example, Figure 8.2 shows the solution for Hello World for this interpreter.

8.1 Exercises

♛'...' 1. Program *anything* (literally anything meaningful will do!) manually directly in the language of TAOTP, i.e. without writing Python - write numerical weights by hand directly.

'...' 2. Make a small change to the TAOTP program from Figure 8.2. Try to predict what will happen. Were you right?

♛ 3. Reverse engineer the TAOTP program from Figure 8.3 - what does it do in general? Try to do it without running it!

'...' 4. Improve the `compile_TAOTP` function from Figure 8.1 to allow more compact inputs. For example, use default values for the layer normalization layers, allow omitting empty `layers` arrays, allow specifying a `block_size` value and automatically initialize `pos_emb` appropriately, etc.

```
{
    "tok_emb": [
        [1.2, -1.2, 0.0],
        [1.4, -0.6, -0.7],
        [1.1, 0.1, -1.2],
        [0.5, 0.8, -1.3],
        [-0.2, 1.3, -1.0],
        [-0.9, 1.3, -0.3],
        [-1.3, 0.9, 0.3],
        [-1.3, 0.2, 1.0],
        [-0.8, -0.5, 1.3],
        [-0.1, -1.1, 1.2],
        [0.6, -1.4, 0.7],
    ],
    "pos_emb": [
        [1412611.7, -648030.1, -764581.5],
        [-874208.8, -525610.0, 1399818.9],
        [1224744.8, -1224744.8, 0.0],
        [1224744.8, -1224744.8, 0.0],
        [-1336432.9, 267641.6, 1068791.2],
        [1151984.3, 134429.5, -1286413.9],
        [-267641.6, 1336432.9, -1068791.2],
        [-1336432.9, 267641.6, 1068791.2],
        [525610.0, 874208.8, -1399818.9],
        [1224744.8, -1224744.8, 0.0],
        [-1374349.0, 975919.0, 398430.0],
        [-975919.0, 1374349.0, -398430.0],
        [648030.1, -1412611.7, 764581.5],
    ],
    "layers": [],
    "lnf": {"gamma": 1.0, "beta": 0.0},
}
```

Figure 8.2: Our solution for Hello World in the format of the TAOTP interpreter.

```
{
    "tok_emb": [
        [1.3, -1.1, -0.2],
        [1.3, -0.9, -0.4],
        [1.4, -0.7, -0.6],
        [1.4, -0.6, -0.7],
        [1.3, -0.4, -0.9],
        [1.3, -0.2, -1.1],
        [1.2, 0, -1.2],
        [1.1, 0.2, -1.3],
        [0.9, 0.4, -1.3],
        [0.7, 0.6, -1.4],
        [0.6, 0.7, -1.4],
        [0.4, 0.9, -1.3],
        [0.2, 1.1, -1.3],
        [0, 1.2, -1.2],
        [-0.2, 1.3, -1.1],
        [-0.4, 1.3, -0.9],
        [-0.6, 1.4, -0.7],
        [-0.7, 1.4, -0.6],
        [-0.9, 1.3, -0.4],
        [-1.1, 1.3, -0.2],
    ],
    "pos_emb": [
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
    ],
    "layers": [
        {
            "Q": [[[70710678.1, 0., 0.], [70710678.1, 0., 0.], [-141421356.2, 0., 0.]]],
            "K": [[[1.2, 0., 0.], [-1.2, 0., 0.], [-0., 0., 0.]]],
            "V": [[[1.0, 0., 0.], [0., 1.0, 0.], [0., 0., 1.0]]],
            "P": [[[1000000., 0., 0.], [0., 1000000., 0.], [0., 0., 1000000.]]],
            "M1": [[], [], []],
            "b1": [],
            "M2": [],
            "b2": [0., 0., 0.],
            "ln1": {"gamma": 1., "beta": 0.},
            "ln2": {"gamma": 1., "beta": 0.},
        }
    ],
    "lnf": {"gamma": 1., "beta": 0.},
}
```

Figure 8.3: Code for Exercise 3.

Index

Bibliography

[Alammar, 2018] Alammar, J. (2018). The illustrated transformer.

[Knuth, 1997a] Knuth, D. E. (1997a). *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., third edition.

[Knuth, 1997b] Knuth, D. E. (1997b). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third edition.

[Knuth, 1998] Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA.

[Press and Wolf, 2017] Press, O. and Wolf, L. (2017). Using the output embedding to improve language models.

[Radford et al., 2018] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training.

[Su et al., 2022] Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., and Liu, Y. (2022). Roformer: Enhanced transformer with rotary position embedding.

[Turner, 2023] Turner, R. E. (2023). An introduction to transformers.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need.

[Xiong et al., 2020] Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., and Liu, T.-Y. (2020). On layer normalization in the transformer architecture.

[Zhang and Sennrich, 2019] Zhang, B. and Sennrich, R. (2019). Root mean square layer normalization.