

First Author

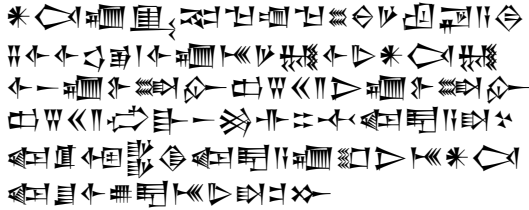
Affiliation / Address line 1
Affiliation / Address line 2
Affiliation / Address line 3
email@domain

Second Author

Affiliation / Address line 1
Affiliation / Address line 2
Affiliation / Address line 3
email@domain

Abstract

This paper announces the discovery of the use of neural nets almost 4,000 years before their use in the modern era. Newly discovered tablets preserve a perceptron used for calculating the numbers on Plimpton 322, the most important object in the history of mathematics. The native programming language used by the ancient Babylonian “cuneogrammers” uses sexagesimal numbering leading to some “weirdness”.



1 Introduction

The history of math is long, but the history of programming is longer. Cuneiform, arguably the first writing in the world, is known for its sheep receipts and beer ration lists as well as complaints about substandard copper (Oppenheim, 1954) and women complaining at each other (Matuszak, 2020). This article adds neural network programming to that vaunted list of human achievements. A set of newly discovered cuneiform tablets preserve the mechanism for performing simple neural network calculations. These methods, it seems, were used to calculate the lengths of triangles; a well known example of this exercise is preserved on the tablet known as Plimpton 322¹. It is remarkable that no historian of math or cuneiform scholar has ever consider this possibility. This paper covers the background, a description of

¹An important and real description of this interesting object is found in Robson (2002).

the cuneogramming language, and includes a facsimile copy of the most important tablet as an appendix.

Unfortunately, the hardware required to execute these programs (i.e. a living Babylonian mathematician) has not been adequately preserved, but we have managed to write a Python library which emulates it.² The assumption is that these calculations were done by hand in their copious free time between inventing the wheel and the concept of zero. While the actual output of these tablets is relatively simple by modern standards, the implications of this discovery are profound. Future work will explore how these techniques could have been used in the realm of astronomical calculation and elucidate the full extent of Babylonian computational prowess.

2 Description of the Language






Programs in 𒂗𒍪𒍪 (EME.ŠID.A “language of counting”) follow a tabular structure with three main sections: (i) a header, denoted by 𒂗𒍪 (DUB “tablet”), (ii) a sequence of instructions, and (iii) a colophon detailing the tablet’s authorship. Each instruction spans four columns, which we have taken to calling the *arguments*, *opcode*, *destination*, and *line number*. These columns are usually tab-separated, though in a few documents they are TAB-separated (using the cuneiform sign TAB 𒀭). Instructions are grouped into blocks by means of horizontal lines.





Arguments An instruction’s arguments may be numbers, register addresses, or a combination of the two. Numbers are encoded following standard Babylonian conventions (?), with 𒌦 denoting the radix point which separates the integer part from a following fraction. There is

²github.com/MrLogarithm/emeszida









an explicit representation for zero (\emptyset), making these tablets some of the earliest unambiguous examples of the mathematical concept of zero.


A register address is denoted by the phrase $\text{NID}_2.\text{KAS}_7\ n\text{-KAM}$ 𐌵𐌹𐌸𐌸𐌹𐌺𐌺𐌹𐌺𐌹 n (‘‘thing.account n -th’’, ‘‘the item in the n -th place’’), where n is any number. 𐌵𐌹𐌸𐌸𐌹𐌺𐌺𐌹𐌺𐌹 expressions can be nested to perform a kind of pointer dereferencing: for example, if 𐌵𐌹𐌸𐌸𐌹𐌺𐌺𐌹𐌺𐌹 (register 1) contains the value 𐌵𐌹𐌸𐌸𐌹𐌺𐌺𐌹 (3), and $\text{𐌵𐌹𐌸𐌸𐌹𐌺𐌺𐌹𐌺𐌹𐌵𐌹𐌸𐌸𐌹𐌺𐌺𐌹}$ (register 3) contains the value 𐌵𐌹𐌸𐌸𐌹𐌺𐌺𐌹𐌵𐌹𐌸𐌸𐌹𐌺𐌺𐌹 (7), then $\text{𐌵𐌹𐌸𐌸𐌹𐌺𐌺𐌹𐌵𐌹𐌸𐌸𐌹𐌺𐌺𐌹𐌵𐌹𐌸𐌸𐌹𐌺𐌺𐌹}$ will evaluate to 7 (the value in the register pointed to by 𐌵𐌹𐌸𐌸𐌹𐌺𐌺𐌹𐌵𐌹𐌸𐌸𐌹𐌺𐌺𐌹).

If an instruction has multiple arguments, these must be delimited by a wide space (distinct from the short space used to separate groups of digits within a number), or by one of the phrases *a-na*  “to” or *i-na*  “from”. By convention, the choice of delimiter depends on the instruction’s opcode (see below), with multiplication operations preferring space delimiters, addition preferring , and subtraction preferring . There is no mechanism to enforce these conventions, but we recommend following them because  is hard enough to read at the best of times.

Destination In most cases, the destination column of an instruction will be a register address where the output is to be stored, e.g.   . Some control-flow instructions (see below) instead expect the destination to be a line number.  SUD (“distant, remote”) can be used as a null destination, for statements which produce no output.

Opcodes Each instruction has a single opcode belonging to the following vocabulary:

-  DAḤ.HA, “add”
-  BA.ZI, “tear out”
-  A.RA , “multiply”
-  IGI, “reciprocal”
-  ME, “to be”
-  *ta-mar*, “you will see”
-  NIGIN.NA, “start again”
-  TUKUM.BI DIRIG, “if it exceeds”





-  TUKUM.BI SIG, “if it it is weak”


The first three of these are binary operators for addition, subtraction, and multiplication respectively. The subtraction operator deserves special attention for its use in constructions of the following shape, which appear hundreds of times throughout the Babylonian programming corpus:

This instruction subtracts $\{\text{I}\}\text{I}\text{I}\text{I}$ from zero and stores the result in $\text{I}\text{I}\text{I}\text{I}\text{I}\text{I}\text{I}\text{I}$. This effectively negates the first argument, and seems to have been the primary way by which Babylonian cuneigrammers represented negative numbers, as their primitive and archaic notation otherwise lacked a means to encode such values. This curious practice gives definitive proof that the invention of negative numbers occurred centuries earlier than previously believed.

The language does not appear to have any kind of binary division operator. Rather, a unary ∇ operator was used to find the reciprocal of the denominator, which was then multiplied by the numerator using the binary \boxtimes operator.

\triangleright is a unary assignment operator which stores a value in a destination register. $\blacktriangleleft\!\!\!\!\!\blacktriangleright\!\!\!\!\!\blacktriangleright\!\!\!\!\!\blacktriangleright$ functions as a unary print operator.

 includes three types of control-flow instructions.  functions like GOTO, and jumps the program counter to the specified line number.  functions like the x86 jz instruction, and jumps to the specified line number if and only if its argument is zero.  is similar, and jumps if the argument is greater than zero.

Line Numbers Every line of an  program ends in a mandatory line number. However, these numbers are not generally sequential, and need not even be distinct. For example, most lines of the perceptron tablet are labeled with the number 0 (zero); only lines that are the destination of some control-flow instruction receive non-zero identifiers.

2.1 Fractional Indexing


Both line numbers and register addresses in  can have fractional parts. The original scribes seem to have exploited this fact



Figure 1: BM 34580, courtesy of the Trustees of the British Museum, CC BY-NC-SA 4.0

to establish non-overlapping “namespaces” for the different parts of their code. For example, in the perceptron tablet, all of the model parameters are stored in addresses with integer part 1; the program inputs all have integer part 2; the matrix multiplication subroutine uses addresses with integer part 3; and so on. The fractional parts of register addresses also appear to follow some standard conventions, with the $X;0$ register typically storing a subroutine’s return address, while $X;1$ onward were used for its arguments.

The perceptron tablet also uses fractional register addresses to perform a kind of multi-dimensional array indexing. As an example, the first layer of the perceptron has a 50×2 weight matrix, and this is stored in registers $1;0,0,0$ through $1;0,49,1$. The integer part of these addresses denotes the “data” portion of memory; the first digit after the radix point identifies this as the 0th model parameter; and the second and third digits can be treated as a pair of indices ranging from 0–49 and 0–1 respectively. To access a specific element in this matrix, the scribes use repeated division by 60 to implement a kind of “bit-shift” instruction, in order to shift integer indices into the correct positions after the radix point. By adding the bit-shifted element indices (e.g. $0;0,4,7$ for the element in row 5, column 8) to a pointer to the top corner of the matrix ($1;0,0,0$) they obtain the address of the desired element ($1;0,4,7$).

Notably, this practice limits the size of their model parameters to at most 60×60 , as for larger values the addresses would carry over to higher digits and thus begin to overwrite

one another. This limitation may explain why AI never made waves in Babylonian society, as their models were all too small to be truly revolutionary.

3 Description of the Texts

This corpus contains numerous fragments implementing recognizable procedures such as bit-shifting, populating an array, computing dot products, and so on. However, only a single text is known to have been preserved in its entirety.³ Spanning close to 1700 lines, this impressive text is divided into five sections implementing what modern readers will immediately recognize to be a multi-layer perceptron. The first section straightforwardly defines a matrix-multiplication subroutine. This is called by a subroutine defined in section two, which applies each layer of the perceptron to a given input, and applies a ReLU-style activation between each pair of layers. Section three implements the tablet’s “main” method, which loads an input to memory, calls the perceptron subroutine, and prints the resulting output. Section four loads the model parameters, which appear to comprise weight matrices of sizes 50×2 , 25×50 , and 1×25 , plus bias vectors of sizes 50, 25, and 1 respectively. The final section lists pairs of inputs, whose values (incredibly!) correspond to the second and third columns of Plimpton 322.

When the program is executed, it produces a single numeric output for each input pair. These outputs correspond remarkably closely

³All of the known texts are reproduced in github.com/MrLogarithm/emeszida/tree/main/programs

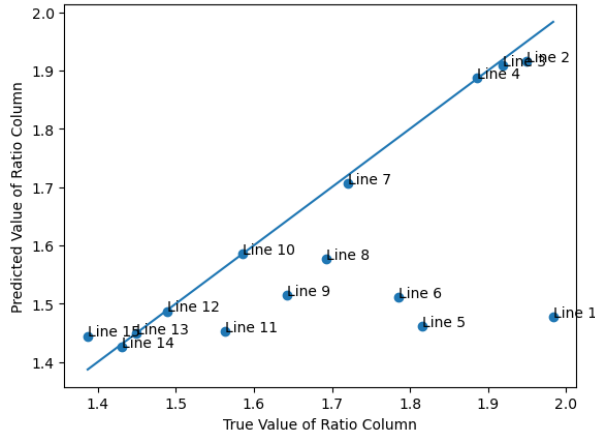


Figure 2: Outputs from the perceptron tablet vs. true value of each line from column 1 of Plimpton 322. If a points falls upon the line, the model output for that line exactly equals the value on Plimpton 322.

to the values in the first column of Plimpton 322, as demonstrated in Figure 2. The correspondence is not perfect, however, and the values which should match lines 1, 5, and 6 of Plimpton 322 are significantly larger than expected. This implies that, although the code on this tablet is clearly *related* to Plimpton 322, it could not have been used to directly populate the table in that text. Perhaps the outputs from this model were refined in some later step to produce the more exact ratios in the Plimpton text, or perhaps the Babylonians were disillusioned by the imprecision of their machine learning models and simply abandoned them for tried-and-true manual methods. Given how miniscule the cuneigramming corpus is relative to the larger body of Babylonian administrative writing, we lean towards the latter explanation.

The tablets preserve a number of methods for performing simple neural network calculations. Simple array operations as well as the calculation of the dot product are included in the corpus. All of this comes together in the large perceptron tablet. It is clear that the cuneogrammers were slowly developing their skills as some of the earlier and simpler methods date as far back as the Old Akkadian period (c. 2300–2150 BCE) while the perceptron tablet is from the Old Babylonian period (c. 1900–1600 BCE). Interestingly, the long weights at the end of the tablet very closely resemble the parameters of later astronomical calculations (see

Figure 1).

4 Implications

This completely rewrites the history of modern computing...

Other ancient corpora which have resisted decipherment, and which boast a similar numeric component, may represent additional examples of ancient programming traditions (Kelley et al., 2022).

- compare the table of params at the end of the file to tables of astronomical parameters from genuine tablets

Acknowledgments

The search for these tablets and our effort to understand them was inspired by Ramsey Nasser’s **قلب** programming language (<https://github.com/nasser/>).

References

- Kathryn Kelley, Logan Born, M. Willis Monroe, and Anoop Sarkar. 2022. [Image-aware language modeling for proto-elamite](#). *Lingue e linguaggio*, (2):261–294.
- Jana Matuszak. 2020. *»Und du, du bist eine Frau?!«: Editio princeps und Analyse des sumerischen Streitgesprächs »Zwei Frauen B«*. De Gruyter.
- A. L. Oppenheim. 1954. [The seafaring merchants of ur](#). *Journal of the American Oriental Society*, 74(1):6–17.
- Eleanor Robson. 2002. [Words and pictures: New light on plimpton 322](#). *The American Mathematical Monthly*, 109(2):105–120.

A Example Appendix

- reproduce entire perceptron-full.eme in the text, without comments:

[illegible]

[illegible][illegible][illegible]

[illegible][illegible]

[illegible][illegible]

[illegible][illegible][illegible]

[illegible][illegible]

[illegible][illegible][illegible]

[The page contains dense, illegible vertical Chinese text.]

[illegible][illegible]

[illegible][illegible][illegible]

[illegible][illegible][illegible]

[illegible][illegible]

𠂇	𠂈	𠂉	𠂊	𠂋	𠂌	𠂍	𠂎	𠂏	𠂐
𠂑	𠂒	𠂓	𠂔	𠂕	𠂖	𠂗	𠂘	𠂙	𠂚
𠂛	𠂜	𠂝	𠂞	𠂟	𠂠	𠂡	𠂢	𠂣	𠂤
𠂥	𠂦	𠂧	𠂨	𠂩	𠂪	𠂫	𠂬	𠂭	𠂮
𠂯	𠂰	𠂱	𠂲	𠂳	𠂴	𠂵	𠂶	𠂷	𠂸
𠂹	𠂺	𠂻	𠂼	𠂽	𠂾	𠂿	𠃀	𠃁	𠃂
𠃃	𠃄	𠃅	𠃆	𠃇	𠃈	𠃉	𠃊	𠃋	𠃌
𠃍	𠃎	𠃏	𠃐	𠃑	𠃒	𠃓	𠃔	𠃕	𠃖
𠃗	𠃘	𠃙	𠃚	𠃛	𠃜	𠃝	𠃞	𠃟	𠃠
𠃡	𠃢	𠃣	𠃤	𠃥	𠃦	𠃧	𠃨	𠃩	𠃪
𠃫	𠃬	𠃭	𠃮	𠃯	𠃰	𠃱	𠃲	𠃳	𠃴
𠃵	𠃶	𠃷	𠃸	𠃹	𠃺	𠃻	𠃼	𠃽	𠃾
𠃿	𠄀	𠄁	𠄂	𠄃	𠄄	𠄅	𠄆	𠄇	𠄈
𠄉	𠄊	𠄋	𠄌	𠄍	𠄎	𠄏	𠄐	𠄑	𠄒
𠄓	𠄔	𠄕	𠄖	𠄗	𠄘	𠄙	𠄚	𠄛	𠄜
𠄝	𠄞	𠄟	𠄠	𠄡	𠄢	𠄣	𠄤	𠄥	𠄦
𠄧	𠄨	𠄩	𠄪	𠄫	𠄬	𠄭	𠄮	𠄯	𠄰
𠄱	𠄲	𠄳	𠄴	𠄵	𠄶	𠄷	𠄸	𠄹	𠄺
𠄻	𠄼	𠄽	𠄾	𠄿	𠅀	𠅁	𠅂	𠅃	𠅄
𠅅	𠅆	𠅇	𠅈	𠅉	𠅊	𠅋	𠅌	𠅍	𠅎
𠅏	𠅐	𠅑	𠅒	𠅓	𠅔	𠅕	𠅖	𠅗	𠅘
𠅙	𠅚	𠅛	𠅜	𠅝	𠅞	𠅟	𠅠	𠅡	𠅢
𠅣	𠅤	𠅥	𠅦	𠅧	𠅨	𠅩	𠅪	𠅫	𠅬
𠅭	𠅮	𠅯	𠅰	𠅱	𠅲	𠅳	𠅴	𠅵	𠅶
𠅷	𠅸	𠅹	𠅺	𠅻	𠅼	𠅽	𠅾	𠅿	𠆀
𠆁	𠆂	𠆃	𠆄	𠆅	𠆆	𠆇	𠆈	𠆉	𠆊
𠆋	𠆌	𠆍	𠆎	𠆏	𠆐	𠆑	𠆒	𠆓	𠆔
𠆕	𠆖	𠆗	𠆘	𠆙	𠆚	𠆛	𠆜	𠆝	𠆞
𠆟	𠆠	𠆡	𠆢	𠆣	𠆤	𠆥	𠆦	𠆧	𠆨
𠆩	𠆪	𠆫	𠆬	𠆭	𠆮	𠆯	𠆰	𠆱	𠆲
𠆳	𠆴	𠆵	𠆶	𠆷	𠆸	𠆹	𠆺	𠆻	𠆼
𠆽	𠆾	𠆿	𠇀	𠇁	𠇂	𠇃	𠇄	𠇅	𠇆
𠇇	𠇈	𠇉	𠇊	𠇋	𠇌	𠇍	𠇎	𠇏	𠇐
𠇑	𠇒	𠇓	𠇔	𠇕	𠇖	𠇗	𠇘	𠇙	𠇚
𠇛	𠇜	𠇝	𠇞	𠇟	𠇠	𠇡	𠇢	𠇣	𠇤
𠇥	𠇦	𠇧	𠇨	𠇩	𠇪	𠇫	𠇬	𠇭	𠇮
𠇯	𠇰	𠇱	𠇲	𠇳	𠇴	𠇵	𠇶	𠇷	𠇸
𠇹	𠇺	𠇻	𠇼	𠇽	𠇾	𠇿	𠈀	𠈁	𠈂
𠈃	𠈄	𠈅	𠈆	𠈇	𠈈	𠈉	𠈊	𠈋	𠈌
𠈍	𠈎	𠈏	𠈐	𠈑	𠈒	𠈓	𠈔	𠈕	𠈖
𠈗	𠈘	𠈙	𠈚	𠈛	𠈜	𠈝	𠈞	𠈟	𠈠
𠈡	𠈢	𠈣	𠈤	𠈥	𠈦	𠈧	𠈨	𠈩	𠈪
𠈫	𠈬	𠈭	𠈮	𠈯	𠈰	𠈱	𠈲	𠈳	𠈴
𠈵	𠈶	𠈷	𠈸	𠈹					

1
 2
 3
 4
 5
 6
 7
 8
 9
 10
 11
 12
 13
 14
 15
 16
 17
 18
 19
 20
 21
 22
 23
 24
 25
 26
 27
 28
 29
 30
 31
 32
 33
 34
 35
 36
 37
 38
 39
 40
 41
 42
 43
 44
 45
 46
 47
 48
 49
 50
 51
 52
 53
 54
 55
 56
 57
 58
 59
 60
 61
 62
 63
 64
 65
 66
 67
 68
 69
 70
 71
 72
 73
 74
 75
 76
 77
 78
 79
 80
 81
 82
 83
 84
 85
 86
 87
 88
 89
 90
 91
 92
 93
 94
 95
 96
 97
 98
 99
 100
 101
 102
 103
 104
 105
 106
 107
 108
 109
 110
 111
 112
 113
 114
 115
 116
 117
 118
 119
 120
 121
 122
 123
 124
 125
 126
 127
 128
 129
 130
 131
 132
 133
 134
 135
 136
 137
 138
 139
 140
 141
 142
 143
 144
 145
 146
 147
 148
 149
 150
 151
 152
 153
 154
 155
 156
 157
 158
 159
 160
 161
 162
 163
 164
 165
 166
 167
 168
 169
 170
 171
 172
 173
 174
 175
 176
 177
 178
 179
 180
 181
 182
 183
 184
 185
 186
 187
 188
 189
 190
 191
 192
 193
 194
 195
 196
 197
 198
 199
 200
 201
 202
 203
 204
 205
 206
 207
 208
 209
 210
 211
 212
 213
 214
 215
 216
 217
 218
 219
 220
 221
 222
 223
 224
 225
 226
 227
 228
 229
 230
 231
 232
 233
 234
 235
 236
 237
 238
 239
 240
 241
 242
 243
 244
 245
 246
 247
 248
 249
 250
 251
 252
 253
 254
 255
 256
 257
 258
 259
 260
 261
 262
 263
 264
 265
 266
 267
 268
 269
 270
 271
 272
 273
 274
 275
 276
 277
 278
 279
 280
 281
 282
 283
 284
 285
 286
 287
 288
 289
 290
 291
 292
 293
 294
 295
 296
 297
 298
 299
 300
 301
 302
 303
 304
 305
 306
 307
 308
 309
 310
 311
 312
 313
 314
 315
 316
 317
 318
 319
 320
 321
 322
 323
 324
 325
 326
 327
 328
 329
 330
 331
 332
 333
 334
 335
 336
 337
 338
 339
 340
 341
 342
 343
 344
 345
 346
 347
 348
 349
 350
 351
 352
 353
 354
 355
 356
 357
 358
 359
 360
 361
 362
 363
 364
 365
 366
 367
 368
 369
 370
 371
 372
 373
 374
 375
 376
 377
 378
 379
 380
 381
 382
 383
 384
 385
 386
 387
 388
 389
 390
 391
 392
 393
 394
 395
 396
 397
 398
 399
 400
 401
 402
 403
 404
 405
 406
 407
 408
 409
 410
 411
 412
 413
 414
 415
 416
 417
 418
 419
 420
 421
 422
 423
 424
 425
 426
 427
 428
 429
 430
 431
 432
 433
 434
 435
 436
 437
 438
 439
 440
 441
 442
 443
 444
 445
 446
 447
 448
 449
 450
 451
 452
 453
 454
 455
 456
 457
 458
 459
 460
 461
 462
 463
 464
 465
 466
 467
 468
 469
 470
 471
 472
 473
 474
 475
 476
 477
 478
 479
 480
 481
 482
 483
 484
 485
 486
 487
 488
 489
 490
 491
 492
 493
 494
 495
 496
 497
 498
 499
 500
 501
 502
 503
 504
 505
 506
 507
 508
 509
 510
 511
 512
 513
 514
 515
 516
 517
 518
 519
 520
 521
 522
 523
 524
 525

[illegible][illegible]

[illegible][illegible][illegible]

[illegible][illegible]

[illegible][illegible][illegible]

[illegible][illegible][illegible][illegible]

