

```

1 # Please do not change this cell because some hidden tests might depend on it.
2 import os
3
4 # Otter grader does not handle ! commands well, so we define and use our
5 # own function to execute shell commands.
6 def shell(commands, warn=True):
7     """Executes the string `commands` as a sequence of shell commands.
8
9     Prints the result to stdout and returns the exit status.
10    Provides a printed warning on non-zero exit status unless `warn`
11    flag is unset.
12    """
13    file = os.popen(commands)
14    print (file.read().rstrip('\n'))
15    exit_status = file.close()
16    if warn and exit_status != None:
17        print(f"Completed with errors. Exit status: {exit_status}\n")
18    return exit_status
19
20 shell("""
21 ls requirements.txt >/dev/null 2>&1
22 if [ ! $? = 0 ]; then
23   rm -rf .tmp
24   git clone https://github.com/cs236299-2022-spring/lab4-5.git .tmp
25   mv .tmp/tests ./
26   mv .tmp/requirements.txt ./
27   rm -rf .tmp
28 fi
29 pip install -q -r requirements.txt
30 """)

```

```

1 # Initialize Otter
2 import otter
3 grader = otter.Notebook()

```

Unsupported Cell Type. Double-Click to inspect/edit the content.

## ▼ Course 236299

### Lab 4-5 - Sequence-to-sequence models with attention

In lab 4-4, you built a sequence-to-sequence model in its most basic form and applied it to the task of words-to-numbers conversion. That model first encodes the source sequence into a fixed-size vector (encoder final states), and then decodes based on that vector. Since the only way information from the source side can flow to the target side is through this fixed-size vector, it presents a bottleneck in the encoder-decoder model: no matter how long the source sentence is, it must always be compressed into this fixed-size vector.

An *attention mechanism* (proposed in [this seminal paper](#)) offers a workaround by providing the decoder a dynamic view of the source-side as the decoding proceeds. Instead of compressing the source sequence into a *fixed-size* vector, we preserve the "resolution" and encode the source sequence into a *set of vectors* (usually with the same size as the source sequence) which is sometimes called a *memory bank*. When predicting each word, the decoder "attends to" this memory bank and assigns a weight to each vector in the set, and the weighted sum of those vectors will be used to make a prediction. Hopefully, the decoder will assign higher weights to more relevant source words when predicting a target word, which we'll test in this lab.

New bits of Pytorch used in this lab, and which you may find useful include:

- [torch.transpose](#): swaps two dimensions of a tensor.
- [torch.reshape](#): reshapes a tensor.
- [torch.bmm](#): Performs batched matrix multiplication.
- [torch.nn.utils.rnn.pack\\_padded\\_sequence](#) (imported as `pack`): Handles paddings. A more detailed explanation can be found [here](#).
- [torch.nn.utils.rnn.pad\\_packed\\_sequence](#) (imported as `unpack`): Handles paddings.
- [torch.masked\\_fill](#): Fills tensor elements with a value in spots where mask is `True`.
- [torch.softmax](#): Computes softmax.
- [torch.repeat](#): Repeats a tensor along the specified dimensions.
- [torch.triu](#): Returns the upper triangular part of a matrix.

# Preparation - Loading data

We use the same data as in lab 4-4.

```
1 import copy
2 import math
3 import matplotlib
4 import matplotlib.pyplot as plt
5 import os
6 import wget
7
8 import torch
9 import torch.nn as nn
10 import torchtext.legacy as tt
11
12 from tqdm import tqdm
13
14 from torch.nn.utils.rnn import pack_padded_sequence as pack
15 from torch.nn.utils.rnn import pad_packed_sequence as unpack
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
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
```

```

19 print(f"Index for start of sequence token: {TGT.vocab.stoi[TGT.init_token]}")
20 print(f"Index for end of sequence token: {TGT.vocab.stoi[TGT.eos_token]}")

```

```

Size of src vocab: 34
Size of tgt vocab: 14
Index for src padding: 1
Index for tgt padding: 1
Index for start of sequence token: 2
Index for end of sequence token: 3

```

We batch training and validation data into minibatches, but for the test set, we use a batch size of 1, to make decoding implementation easier.

```

1 BATCH_SIZE = 32      # batch size for training and validation
2 TEST_BATCH_SIZE = 1 # batch size for test; we use 1 to make implementation easier
3
4 train_iter, val_iter = tt.data.BucketIterator.splits((train_data, val_data),
5                                                     batch_size=BATCH_SIZE,
6                                                     device=device,
7                                                     repeat=False,
8                                                     sort_key=lambda x: len(x.src), # sort by length to minimize padding
9                                                     sort_within_batch=True)
10 test_iter = tt.data.BucketIterator(test_data,
11                                   batch_size=TEST_BATCH_SIZE,
12                                   device=device,
13                                   repeat=False,
14                                   sort=False,
15                                   train=False)

```

Let's take a look at a batch from these iterators.

```

1 batch = next(iter(train_iter))
2 src, src_lengths = batch.src
3 print (f"Size of src batch: {src.shape}")
4 print (f"Third src sentence in batch: {src[:, 2]}")
5 print (f"Length of the third src sentence in batch: {src_lengths[2]}")
6 print (f"Converted back to string: {' '.join([SRC.vocab.itos[i] for i in src[:, 2]])}")
7
8 tgt = batch.tgt
9 print (f"Size of tgt batch: {tgt.shape}")
10 print (f"Third tgt sentence in batch: {tgt[:, 2]}")
11 print (f"Converted back to string: {' '.join([TGT.vocab.itos[i] for i in tgt[:, 2]])}")

```

```

Size of src batch: torch.Size([13, 32])
Third src sentence in batch: tensor([ 9,  5,  9,  3,  2, 15, 14,  4, 10,  3,  2, 15,  9], device='cuda:0')
Length of the third src sentence in batch: 13
Converted back to string: three million three hundred and thirty four thousand nine hundred and thirty three
Size of tgt batch: torch.Size([12, 32])
Third tgt sentence in batch: tensor([ 2,  4,  4,  4, 10,  5,  4,  4,  3,  1,  1,  1], device='cuda:0')
Converted back to string: <bos> 3 3 3 4 9 3 3 <eos> <pad> <pad> <pad>

```

## ▼ The attention mechanism

Attention works by *querying* a (dynamically sized) set of *keys* associated with *values*. As usual, the query, keys, and values are represented as vectors. The query process provides a score that specifies how much each key should be attended to. The attention can then be summarized by taking an average of the values weighted by the attention score of the corresponding keys. This *context vector* can then be used as another input to other processes.

More formally, let's suppose we have a query vector  $\mathbf{q} \in \mathbb{R}^D$ , a set of  $S$  key-value pairs  $\{(\mathbf{k}_i, \mathbf{v}_i) \in \mathbb{R}^D \times \mathbb{R}^D : i \in \{1, 2, \dots, S\}\}$ , where  $D$  is the hidden size. What we want to do through the attention mechanism is to use the query to attend to the keys, and summarize those values associated with the "relevant" keys into a fixed-size context vector  $\mathbf{c} \in \mathbb{R}^D$ . Note that this is different from directly compressing the key-value pairs into a fixed-size vector, since depending on the query, we might end up with different context vectors.

To determine the score for a given query and key, it is standard to use a measure of similarity between the query and key. You've seen such similarity measures before, in labs 1-1 and 1-2. A good choice is simply the normalized dot product between query and key. We'll thus take the attention score for query  $\mathbf{q}$  and key  $\mathbf{k}_i$  to be

$$a_i = \frac{\exp(\mathbf{q} \cdot \mathbf{k}_i)}{Z},$$

where  $\cdot$  denotes the dot product (inner product) and  $\exp$  is exponentiation which ensures that all scores are nonnegative, and

$$Z = \sum_{i=1}^S \exp(\mathbf{q} \cdot \mathbf{k}_i)$$

is the normalizer to guarantee the scores all sum to one. (There are multiple ways of parameterizing the attention function, but the form we present here is the most popular one.) You might have noticed that the operation above is essentially a softmax over  $\mathbf{q} \cdot \mathbf{k}$ .

The attention scores  $\mathbf{a}$  lie on a *simplex* (meaning  $a_i \geq 0$  and  $\sum_i a_i = 1$ ), which lends it some interpretability: the closer  $a_i$  is to 1, the more "relevant" a key  $k_i$  (and hence its value  $v_i$ ) is to the given query. We will observe this later in the lab: When we are about to predict the target

word "3",  $a_i$  is close to 1 for the source word  $x_i = \text{"three"}$ .

To compute the context vector  $\mathbf{c}$ , we take the weighted sum of values using the corresponding attention scores as weights:

$$\mathbf{c} = \sum_{i=1}^S a_i \mathbf{v}_i$$

The closer  $a_i$  is to 1, the higher the weight  $\mathbf{v}_i$  receives.

---

**Question:** In the extreme, if there exists  $i$  for which  $a_i$  is 1, then what will the value of  $\mathbf{c}$  be?

**In this particular case for every  $j \neq i$   $a_j=0$ , and thus  $\mathbf{c}=\mathbf{v}_i$ .**

In practice, instead of computing the context vector once for each query, we want to batch computations for different queries together for parallel processing on GPUs. This will become especially useful for the transformer implementation. We use a matrix  $Q \in \mathbb{R}^{T \times D}$  to store  $T$  queries, a matrix  $K \in \mathbb{R}^{S \times D}$  to store  $S$  keys, and a matrix  $V \in \mathbb{R}^{S \times D}$  to store the corresponding values. Then we can write down how we compute the attention scores  $A \in \mathbb{R}^{T \times S}$  in a matrix form:

$$A = \text{softmax}(QK^\top, \text{dim} = -1),$$

---

**Question:** What is the shape of  $A$ ? What does  $A_{ij}$  represent?

**The shape of  $A$  is  $T \times S$ .  $A_{ij}$  represent the normalized weight  $a_{ij}$  which is the "relevant score" of query  $i$  to key  $j$  (according to the reading materials, chapter 9).**

To get the context matrix  $C \in \mathbb{R}^{T \times D}$ :

$$C = AV$$

Your first job is to implement this calculation by finishing the attention function below, which takes the  $Q$ ,  $K$ , and  $V$  matrices and returns the  $A$  and  $C$  matrices. Note that for these matrices, there is one additional dimension for the batching, so instead of  $Q \in \mathbb{R}^{T \times D}$ ,  $K, V \in \mathbb{R}^{S \times D}$ ,  $A \in \mathbb{R}^{T \times S}$ ,  $C \in \mathbb{R}^{T \times D}$ , we have  $Q \in \mathbb{R}^{T \times B \times D}$ ,  $K, V \in \mathbb{R}^{S \times B \times D}$ ,  $A \in \mathbb{R}^{B \times T \times S}$ ,  $C \in \mathbb{R}^{T \times B \times D}$ , where  $B$  is the batch size. In addition, the function below also takes an argument `mask` of size  $\mathbb{R}^{B \times T \times S}$  to mark where attentions are disallowed. This is useful not only in disallowing attending to padding symbols, but also in implementing the transformer model which we'll see later in this lab.

**Hint:** Notice that the batch dimension is the second dimension in  $Q$ ,  $K$ ,  $V$ , and  $C$ , but it is the first dimension in  $A$  and `mask`.

**Hint:** You might find [torch.bmm](#) helpful for batched matrix multiplications. You might need to transpose and reshape tensors to be able to use this function.

**Hint:** As mentioned in the beginning of the lab, you might also find [torch.transpose](#), [torch.reshape](#), [torch.masked\\_fill](#), and [torch.softmax](#) useful.

**Hint:** A simple trick for masking an attention score is to set it to negative infinity before normalization.

```
1 #TODO - finish implementing this function.
2 def attention(batched_Q, batched_K, batched_V, mask=None):
3     """
4     Performs the attention operation and returns the attention matrix
5     `batched_A` and the context matrix `batched_C` using queries
6     `batched_Q`, keys `batched_K`, and values `batched_V`.
7
8     Arguments:
9         batched_Q: (q_len, bsz, D)
10        batched_K: (k_len, bsz, D)
11        batched_V: (k_len, bsz, D)
12        mask: (bsz, q_len, k_len). An optional boolean mask *disallowing*
13            attentions where the mask value is *False*.
14
15    Returns:
16        batched_A: the normalized attention scores (bsz, q_len, k_len)
17        batched_C: a tensor of size (q_len, bsz, D).
18    """
19    # Check sizes
20    D = batched_Q.size(-1)
21    bsz = batched_Q.size(1)
22    q_len = batched_Q.size(0)
23    k_len = batched_K.size(0)
24    assert batched_K.size(-1) == D and batched_V.size(-1) == D
25    assert batched_K.size(1) == bsz and batched_V.size(1) == bsz
26    assert batched_V.size(0) == k_len
27
28    if mask is not None:
29        assert mask.size() == torch.Size([bsz, q_len, k_len])
```

```

29     assert mask.size() == torch.Size([bsz, q_len, k_len])
30     batched_Q = batched_Q.transpose(0,1)
31     batched_K = batched_K.transpose(0,1)
32     batched_K = batched_K.transpose(1,2)
33     A = torch.bmm(batched_Q, batched_K)
34     if mask is not None:
35         A = A.masked_fill(~mask, float('-inf'))
36     batched_A = torch.softmax(A, dim=-1)
37     batched_V = batched_V.transpose(0,1)
38     batched_C = torch.bmm(batched_A, batched_V)
39     batched_C = batched_C.transpose(0,1)
40     # Verify that things sum up to one properly.
41     assert torch.all(torch.isclose(batched_A.sum(-1),
42                                     torch.ones(bsz, q_len).to(device)))
43     return batched_A, batched_C

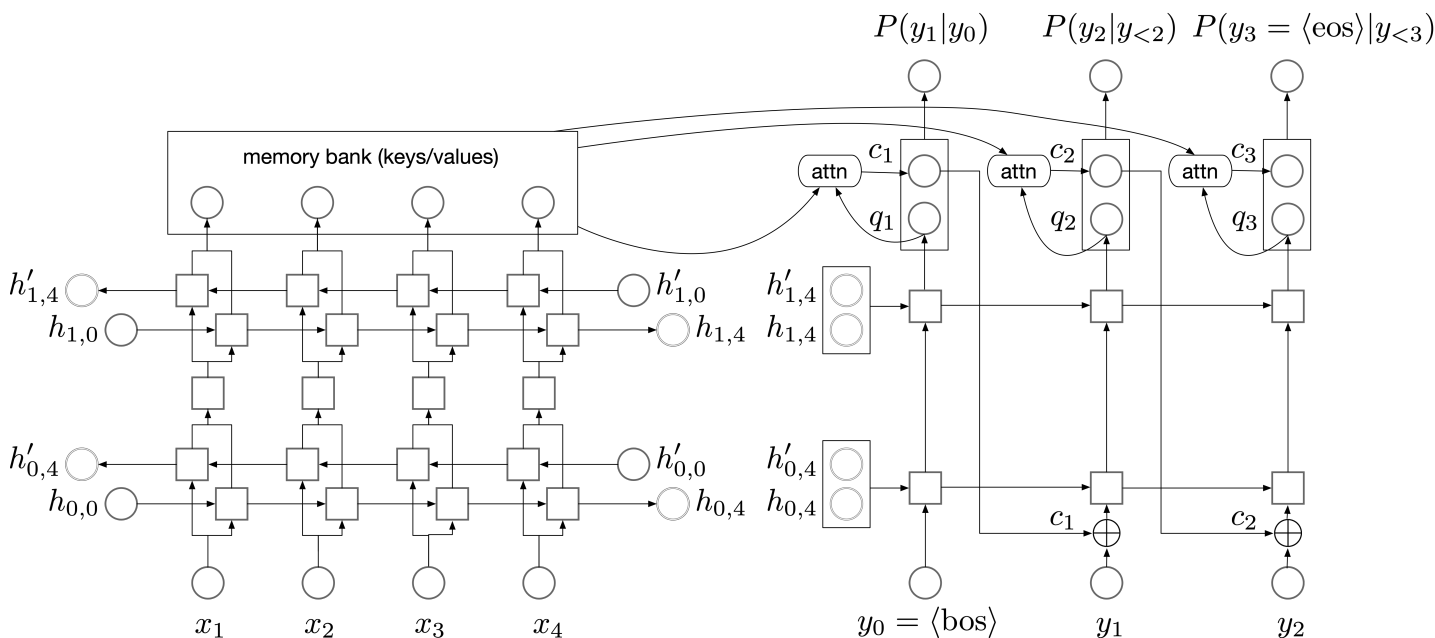
```

```
1 grader.check("attention")
```

All tests passed!

## Neural encoder-decoder models with attention

Now we can add an attention mechanism to our encoder-decoder model. As in lab 4-4, we use a bidirectional LSTM as the encoder, and a unidirectional LSTM as the decoder, and initialize the decoder state with the encoder final state. However, instead of directly projecting the decoder hidden state to logits, we use it as a query vector and attend to all encoder outputs (used as both keys and values), and then concatenate the resulting context vector with the query vector, and project to logits. In addition, we add the context vector to the word embedding at the next time step, so that the LSTM can be aware of the previous attention results.



In the above illustration, at the first time step, we use  $q_1$  to denote the decoder output. Instead of directly projecting that to logits as in lab 4-4, we use  $q_1$  as the query vector, and use it to attend to the memory bank (which is the set of encoder outputs) and get the context vector  $c_1$ . We concatenate  $c_1$  with  $q_1$ , and project the result to the vocabulary size to get logits. At the next step, we first embed  $y_1$  into embeddings, and then **add**  $c_1$  to it (via componentwise addition) and use the sum as the decoder input. This process continues until an end-of-sequence is produced.

You'll need to implement `forward_encoder` and `forward_decoder_incrementally` in the code below. The `forward_encoder` function will return a "memory bank" in addition to the final states. The "memory bank" is simply the encoder outputs at all time steps, which is the first returned value of `torch.nn.LSTM`.

The `forward_decoder_incrementally` function forwards the LSTM cell for a single time step. It takes the initial decoder state, the memory bank, and the input word at the current time step and returns logits for this time step. In addition, it needs to return the context vector and the updated decoder state, which will be used for the next time step. Note that here you need to consider **batch sizes greater than 1**, as this function is used in `forward_decoder`, which is used during training.

In summary, the steps in decoding are:

1. Map the target words to word embeddings. Add the context vector from the previous time step if any. Use the result as the input to the decoder.
2. Forward the decoder RNN for one time step. Use the decoder output as query, the memory bank as **both keys and values**, and compute the context vector through the attention mechanism. Since we don't want to attend to padding symbols at the source side, we also need to pass in a proper `mask` to the attention function.

3. Concatenate the context vector with the decoder output, and project the concatenation to vocabulary size as (unnormalized) logits.

Normalize them using `torch.log_softmax` if `normalize` is `True`.

4. Update the decoder hidden state and the context vector, which will be used in the next time step.

Before proceeding, let's consider a simple question: in lab 4-4, we tried to avoid `for` loops, but if you read the code of `forward_decoder` in this

**Question:** Recall that in the `forward_decoder` function in lab 4-4 we didn't use any `for` loops but instead used a single call to `self.decoder_rnn`. Why do we need a `for` loop in the function `forward_decoder` below? Is it possible to get rid of the `for` loop to make the code more efficient?

**The need of the `for` loop comes from our usage of the attention-context.**

**We note that during decoding, in each iteration we pass in to `decoder_incrementally` function `target[i]`, memory bank and the previous `decoder_state`. The previous `decoder_state` depends on the computed attention-context of that time-step, which depends on the output of the RNN for that time-step. Meaning, if our `decoder-state` wouldn't be depend on the context, or if our context wouldn't be depend on the RNN output (but on the input alone), we could've accelerated it without the `for` loop.**

**However, according to what we stated above, we can not get rid of the `for` loop in this case.**

Now let's implement `forward_encoder` and `forward_decoder_incrementally`.

Hint on using `pack`: if you use `pack` to handle paddings and pass the result as encoder inputs, you need to use `unpack` and extract the first returned value as the memory bank. An example can be found [here](#), but note that our input is already the padded sequences, and that we set `batch_first` to `False`. Hint on ignoring source-side paddings in the attention mechanism: what `mask` should we pass into the `attention` function??

```
1 #TODO - implement `forward_encoder` and `forward_decoder_incrementally`.
2 class AttnEncoderDecoder(nn.Module):
3     def __init__(self, src_field, tgt_field, hidden_size=64, layers=3):
4         """
5         Initializer. Creates network modules and loss function.
6         Arguments:
7             src_field: src field
8             tgt_field: tgt field
9             hidden_size: hidden layer size of both encoder and decoder
10            layers: number of layers of both encoder and decoder
11        """
12        super().__init__()
13        self.src_field = src_field
14        self.tgt_field = tgt_field
15
16        # Keep the vocabulary sizes available
17        self.V_src = len(src_field.vocab.itos)
18        self.V_tgt = len(tgt_field.vocab.itos)
19
20        # Get special word ids
21        self.padding_id_src = src_field.vocab.stoi[src_field.pad_token]
22        self.padding_id_tgt = tgt_field.vocab.stoi[tgt_field.pad_token]
23        self.bos_id = tgt_field.vocab.stoi[tgt_field.init_token]
24        self.eos_id = tgt_field.vocab.stoi[tgt_field.eos_token]
25
26        # Keep hyper-parameters available
27        self.embedding_size = hidden_size
28        self.hidden_size = hidden_size
29        self.layers = layers
30
31        # Create essential modules
32        self.word_embeddings_src = nn.Embedding(self.V_src, self.embedding_size)
33        self.word_embeddings_tgt = nn.Embedding(self.V_tgt, self.embedding_size)
34
35        # RNN cells
36        self.encoder_rnn = nn.LSTM(
37            input_size = self.embedding_size,
38            hidden_size = hidden_size // 2, # to match decoder hidden size
39            num_layers = layers,
40            bidirectional = True # bidirectional encoder
41        )
42        self.decoder_rnn = nn.LSTM(
43            input_size = self.embedding_size,
44            hidden_size = hidden_size,
45            num_layers = layers,
46            bidirectional = False # unidirectional decoder
47        )
48
49        # Final projection layer
50        self.hidden2output = nn.Linear(2*hidden_size, self.V_tgt) # project the concatenation to logits
51
52        # Create loss function
```

[illegible]

```

137 Forward the decoder with token `tgt_in_onestep`.
138 This function will be used both in `forward_decoder` and in beam search.
139 Note that bsz can be greater than 1.
140 Arguments:
141     prev_decoder_states: a tuple (prev_decoder_state, prev_context). `prev_context`
142                          is `None` for the first step
143     tgt_in_onestep: a tensor of size (bsz), tokens at one step
144     memory_bank: a tensor of size (src_len, bsz, hidden_size), encoder outputs
145                  at every position
146     src_mask: a tensor of size (src_len, bsz): a boolean tensor, `False` where
147               src is padding (we disallow decoder to attend to those places).
148     normalize: use log_softmax to normalize or not. Beam search needs to normalize,
149               while `forward_decoder` does not
150 Returns:
151     logits: log probabilities for `tgt_in_token` of size (bsz, V_tgt)
152     decoder_states: (`decoder_state`, `context`) which will be used for the
153                    next incremental update
154     attn: normalized attention scores at this step (bsz, src_len)
155 """
156 prev_decoder_state, prev_context = prev_decoder_states
157 #TODO
158 # ours VVV
159 tgt_embeddings = self.word_embeddings_tgt(tgt_in_onestep[None,...])
160 # Forward decoder RNN
161 input_decoder = tgt_embeddings + (prev_context if prev_context is not None else 0)
162 decoder_outs, decoder_state = self.decoder_rnn(input_decoder, prev_decoder_state)
163
164 src_mask = src_mask.transpose(0,1)
165 src_mask = torch.unsqueeze(src_mask,1)
166 attn, attn_context = attention(decoder_outs, memory_bank, memory_bank, src_mask)
167 concated = torch.cat((decoder_outs, attn_context),dim=2)
168 logits = self.hidden2output(concated)
169 # ours ^^^
170 decoder_states = (decoder_state, attn_context)
171 if normalize:
172     logits = torch.log_softmax(logits, dim=-1)
173 return logits, decoder_states, attn
174
175 def evaluate_ppl(self, iterator):
176     """Returns the model's perplexity on a given dataset `iterator`."""
177     # Switch to eval mode
178     self.eval()
179     total_loss = 0
180     total_words = 0
181     for batch in iterator:
182         # Input and target
183         src, src_lengths = batch.src
184         tgt = batch.tgt # max_length_sql, bsz
185         tgt_in = tgt[:-1] # remove <eos> for decode input (y_0=<bos>, y_1, y_2)
186         tgt_out = tgt[1:] # remove <bos> as target (y_1, y_2, y_3=<eos>)
187         # Forward to get logits
188         logits = self.forward(src, src_lengths, tgt_in)
189         # Compute cross entropy loss
190         loss = self.loss_function(logits.view(-1, self.V_tgt), tgt_out.view(-1))
191         total_loss += loss.item()
192         total_words += tgt_out.ne(self.padding_id_tgt).float().sum().item()
193     return math.exp(total_loss/total_words)
194
195 def train_all(self, train_iter, val_iter, epochs=10, learning_rate=0.001):
196     """Train the model."""
197     # Switch the module to training mode
198     self.train()
199     # Use Adam to optimize the parameters
200     optim = torch.optim.Adam(self.parameters(), lr=learning_rate)
201     best_validation_ppl = float('inf')
202     best_model = None
203     # Run the optimization for multiple epochs
204     for epoch in range(epochs):
205         total_words = 0
206         total_loss = 0.0
207         for batch in tqdm(train_iter):
208             # Zero the parameter gradients
209             self.zero_grad()
210             # Input and target
211             src, src_lengths = batch.src # text: max_src_length, bsz
212             tgt = batch.tgt # max_tgt_length, bsz
213             tgt_in = tgt[:-1] # Remove <eos> for decode input (y_0=<bos>, y_1, y_2)
214             tgt_out = tgt[1:] # Remove <bos> as target (y_1, y_2, y_3=<eos>)
215             bsz = tgt.size(1)
216             # Run forward pass and compute loss along the way.
217             logits = self.forward(src, src_lengths, tgt_in)
218             loss = self.loss_function(logits.view(-1, self.V_tgt), tgt_out.view(-1))
219             # Training stats
220             num_tgt_words = tgt_out.ne(self.padding_id_tgt).float().sum().item()

```



```

221         total_words += num_tgt_words
222         total_loss += loss.item()
223         # Perform backpropagation
224         loss.div(bsz).backward()
225         optim.step()
226
227     # Evaluate and track improvements on the validation dataset
228     validation_ppl = self.evaluate_ppl(val_iter)
229     self.train()
230     if validation_ppl < best_validation_ppl:
231         best_validation_ppl = validation_ppl
232         self.best_model = copy.deepcopy(self.state_dict())
233     epoch_loss = total_loss / total_words
234     print (f'Epoch: {epoch} Training Perplexity: {math.exp(epoch_loss):.4f} '
235           f'Validation Perplexity: {validation_ppl:.4f}')

```

```

1 EPOCHS = 2 # epochs, we highly recommend starting with a smaller number like 1
2 LEARNING_RATE = 2e-3 # learning rate
3
4 # Instantiate and train classifier
5 model = AttnEncoderDecoder(SRC, TGT,
6     hidden_size    = 64,
7     layers         = 3,
8 ).to(device)
9
10 model.train_all(train_iter, val_iter, epochs=EPOCHS, learning_rate=LEARNING_RATE)
11 model.load_state_dict(model.best_model)

```

```

100%|██████████| 2032/2032 [00:53<00:00, 38.21it/s]
Epoch: 0 Training Perplexity: 1.3616 Validation Perplexity: 1.0049
100%|██████████| 2032/2032 [00:49<00:00, 41.35it/s]
Epoch: 1 Training Perplexity: 1.0359 Validation Perplexity: 1.0046
<All keys matched successfully>

```

Since the task we consider here is very simple, we should expect a perplexity very close to 1.

```

1 # Evaluate model performance, the expected value should be < 1.05
2 print (f'Test perplexity: {model.evaluate_ppl(test_iter):.3f}')

```

```
Test perplexity: 1.005
```

```
1 grader.check("encoder_decoder_ppl")
```

```
All tests passed!
```

## Beam search decoding

We can reuse most of our beam search code in lab 4-4 here: we only need to modify the code a bit to pass in `memory_bank` and `src_mask`. For reference here is the same pseudo-code used in lab 4-4, where we want to decode a single example `x` of maximum length `max_T` using a beam size of `K`.

```

1. def beam_search(x, K, max_T):
2.     finished = []          # for storing completed hypotheses
3.     # Initialize the beam
4.     beams = [Beam(hyp=(bos), score=0)] # initial hypothesis: bos, initial score: 0
5.
6.     for t in [1..max_T] # main body of search over time steps
7.         hypotheses = []
8.
9.         # Expand each beam by all possible tokens y_{t+1}
10.        for beam in beams:
11.            y_{1:t}, score = beam.hyp, beam.score
12.            for y_{t+1} in V:
13.                y_{1:t+1} = y_{1:t} + [y_{t+1}]
14.                new_score = score + log P(y_{t+1} | y_{1:t}, x)
15.                hypotheses.append(Beam(hyp=y_{1:t+1}, score=new_score))
16.
17.        # Find K best next beams
18.        beams = sorted(hypotheses, key=lambda beam: -beam.score)[:K]
19.
20.        # Set aside finished beams (those that end in <eos>)
21.        for beam in beams:
22.            y_{t+1} = beam.hyp[-1]
23.            if y_{t+1} == eos:

```

```

16.         finished.append(beam)
17.         beams.remove(beam)

        # Break the loop if everything is finished
18.         if len(beams) == 0:
19.             break
20.     return sorted(finished, key=lambda beam: -beam.score)[0] # return the best finished hypothesis

```

Implement function `beam_search` in the code below. In addition to the predicted target sequence, this function also returns a list of attentions `all_attns`.

```

1 # max target length
2 MAX_T = 15
3 class Beam():
4     """
5     Helper class for storing a hypothesis, its score and its decoder hidden state.
6     """
7     def __init__(self, decoder_state, tokens, score):
8         self.decoder_state = decoder_state
9         self.tokens = tokens
10        self.score = score
11
12 class BeamSearcher():
13     """
14     Main class for beam search.
15     """
16     def __init__(self, model):
17         self.model = model
18         self.bos_id = model.bos_id
19         self.eos_id = model.eos_id
20         self.padding_id_src = model.padding_id_src
21         self.V = model.V_tgt
22
23
24     def beam_search(self, src, src_lengths, K, max_T=MAX_T):
25         """
26         Performs beam search decoding.
27         Arguments:
28             src: src batch of size (max_src_len, 1)
29             src_lengths: src lengths of size (1)
30             K: beam size
31             max_T: max possible target length considered
32         Returns:
33             a list of token ids and a list of attentions
34         """
35         finished = []
36         all_attns = []
37         # Initialize the beam
38         self.model.eval()
39         #TODO - fill in `memory_bank`, `encoder_final_state`, and `init_beam` below
40         # ours VVV
41         memory_bank, encoder_final_state = self.model.forward_encoder(src, src_lengths)
42         init_beam = Beam(encoder_final_state, [torch.LongTensor(1).fill_(self.bos_id).to(device)], score=0)
43         # ours ^^^
44         beams = [init_beam]
45
46         with torch.no_grad():
47             for t in range(max_T): # main body of search over time steps
48
49                 # Expand each beam by all possible tokens y_{t+1}
50                 all_total_scores = []
51                 for beam in beams:
52                     y_1_to_t, score, decoder_state = beam.tokens, beam.score, beam.decoder_state
53                     y_t = y_1_to_t[-1]
54                     #TODO - finish the code below
55                     # Hint: you might want to use `model.forward_decoder_incrementally` with `normalize=True`
56                     src_mask = src.ne(self.padding_id_src)
57                     # ours: VVV
58                     y_t_tensor = torch.ones(1, dtype=torch.long, device=device) * y_t
59                     logits, decoder_state, attn = self.model.forward_decoder_incrementally(decoder_state,
60                                                                                           y_t_tensor, memory_bank, src_mask, normalize=True)
61
62                     if attn is not None:
63                         attn = attn.reshape(1, -1)
64                         total_scores = score + logits
65                         # ours ^^^
66                         all_total_scores.append(total_scores)
67                         all_attns.append(attn) # keep attentions for visualization
68                         beam.decoder_state = decoder_state # update decoder state in the beam
69                 all_total_scores = torch.stack(all_total_scores) # (K, V) when t>0, (1, V) when t=0

```

```

70     # Find K best next beams
71     # The code below has the same functionality as line 6-12, but is more efficient
72     all_scores_flattened = all_total_scores.view(-1) # K*V when t>0, 1*V when t=0
73     topk_scores, topk_ids = all_scores_flattened.topk(K, 0)
74     beam_ids = topk_ids.div(self.V, rounding_mode='floor')
75     next_tokens = topk_ids - beam_ids * self.V
76     new_beams = []
77     for k in range(K):
78         beam_id = beam_ids[k] # which beam it comes from
79         y_t_plus_1 = next_tokens[k] # which y_{t+1}
80         score = topk_scores[k]
81         beam = beams[beam_id]
82         decoder_state = beam.decoder_state
83         y_1_to_t = beam.tokens
84         #TODO
85         new_beam = Beam(decoder_state, y_1_to_t + [y_t_plus_1], score) # ours
86         new_beams.append(new_beam)
87     beams = new_beams
88
89
90     # Set aside completed beams
91     # TODO - move completed beams to `finished` (and remove them from `beams`)
92
93     # ours VV
94     new_beams = []
95     for beam in beams:
96         if beam.tokens[-1] == self.eos_id:
97             finished.append(beam)
98         else:
99             new_beams.append(beam)
100     beams = new_beams
101     # ours ^^^
102
103     # Break the loop if everything is completed
104     if len(beams) == 0:
105         break
106
107     # Return the best hypothesis
108     if len(finished) > 0:
109         finished = sorted(finished, key=lambda beam: -beam.score)
110         return finished[0].tokens, all_attns
111     else: # when nothing is finished, return an unfinished hypothesis
112         return beams[0].tokens, all_attns

```

```
1 grader.check("beam_search")
```

All tests passed!

Now we can use beam search decoding to predict the outputs for the test set inputs using the trained model. You should expect an accuracy close to 100%.

```

1 DEBUG_FIRST = 10 # set to 0 to disable printing predictions
2 K = 1 # beam size 1
3
4 correct = 0
5 total = 0
6
7 # create beam searcher
8 beam_searcher = BeamSearcher(model)
9
10 for index, batch in enumerate(test_iter, start=1):
11     # Input and output
12     src, src_lengths = batch.src
13     # Predict
14     prediction, _ = beam_searcher.beam_search(src, src_lengths, K)
15     # Convert to string
16     prediction = ' '.join([TGT.vocab.itos[token] for token in prediction])
17     prediction = prediction.lstrip('<bos>').rstrip('<eos>').strip()
18     ground_truth = ' '.join([TGT.vocab.itos[token] for token in batch.tgt.view(-1)])
19     ground_truth = ground_truth.lstrip('<bos>').rstrip('<eos>').strip()
20     if DEBUG_FIRST > index:
21         src = ' '.join([SRC.vocab.itos[item] for item in src.view(-1)])
22         print(f'Source: {src}')
23         print(f'Prediction: {prediction}')
24         print(f'Ground truth: {ground_truth}')
25     if ground_truth == prediction:
26         correct += 1
27     total += 1
28
29 print(f'Accuracy: {correct/total:.2f}')

```

Source: sixteen thousand eight hundred and thirty two  
Prediction: 1 6 8 3 2  
Ground truth: 1 6 8 3 2  
Source: sixty seven million six hundred and eighty five thousand two hundred and thirty  
Prediction: 6 7 6 8 5 2 3 0  
Ground truth: 6 7 6 8 5 2 3 0  
Source: six thousand two hundred and twelve  
Prediction: 6 2 1 2  
Ground truth: 6 2 1 2  
Source: seven hundred and ninety eight million three hundred and thirty one thousand eight hundred and eighteen  
Prediction: 7 9 8 3 3 1 8 1 8  
Ground truth: 7 9 8 3 3 1 8 1 8  
Source: eighty eight million four hundred and thirteen thousand nine hundred and eighteen  
Prediction: 8 8 4 1 3 9 1 8  
Ground truth: 8 8 4 1 3 9 1 8  
Source: three hundred and seventy four thousand two hundred and seventy  
Prediction: 3 7 4 2 7 0  
Ground truth: 3 7 4 2 7 0  
Source: ninety eight million three hundred and seventy thousand five hundred and forty five  
Prediction: 9 8 3 7 0 5 4 5  
Ground truth: 9 8 3 7 0 5 4 5  
Source: ninety seven thousand seven hundred and sixty two  
Prediction: 9 7 7 6 2  
Ground truth: 9 7 7 6 2  
Source: four hundred and ten thousand two hundred and three  
Prediction: 4 1 0 2 0 3  
Ground truth: 4 1 0 2 0 3  
Accuracy: 1.00

## Visualizing attention

We can visualize how each query distributes its attention scores over each source word.

```
1 K = 1 # this code only works for beam size 1
2
3 # Create beam searcher
4 beam_searcher = BeamSearcher(model)
5 batch = next(iter(test_iter))
6 # Input and output
7 src, src_lengths = batch.src
8 # Predict and get attentions
9 prediction, all_attns = beam_searcher.beam_search(src, src_lengths, K)
10 all_attns = torch.stack(all_attns, 0)
11 # Convert to string
12 prediction = ' '.join([TGT.vocab.itos[token] for token in prediction])
13 prediction = prediction.lstrip('<bos>').rstrip('<eos>').strip()
14 ground_truth = ' '.join([TGT.vocab.itos[token] for token in batch.tgt.view(-1)])
15 ground_truth = ground_truth.lstrip('<bos>').rstrip('<eos>').strip()
16 src = ' '.join([SRC.vocab.itos[item] for item in src.view(-1)])
17 print (f'Source: {src}')
18 print (f'Prediction: {prediction}')
19 print (f'Ground truth: {ground_truth}')
20
21 # Plot
22 fig, ax = plt.subplots(figsize=(8, 6))
23
24 ax.imshow(all_attns[:,0,:].detach().cpu())
25 ax.set_yticks(list(range(1+len(prediction.split()))));
26 ax.set_yticklabels(prediction.split() + ['eos']);
27 ax.set_xticks(list(range(len(src.split()))));
28 ax.set_xticklabels(src.split());
29
30 # Uncomment the line below if the plot does not show up
31 # Make sure to comment that before submitting to gradescope
32 # since there would be some autograder issues with plt.show()
33 #plt.show()
```

Source: sixteen thousand eight hundred and thirty two  
 Prediction: 1 6 8 3 2  
 Ground truth: 1 6 8 3 2



Do these attentions make sense? Do you see how the attention mechanism solves the bottleneck problem in vanilla seq2seq?



## ▼ The transformer architecture



In RNN-based neural encoder-decoder models, we used recurrence to model the dependencies among words. For example, by running a unidirectional RNN from  $y_1$  to  $y_t$ , we can consider the past history when predicting  $y_{t+1}$ . However, running an RNN over a sequence is a serial process: we need to wait for it to finish running from  $y_1$  to  $y_t$  before being able to compute the outputs at  $y_{t+1}$ . This serial process cannot be parallelized on GPUs along the sequence length dimension: even during training where all  $y_t$ 's are available, we cannot compute the logits for  $y_t$  and the logits for  $y_{t+1}$  in parallel.

The attention mechanism provides an alternative, and most importantly, parallelizable solution. [The transformer model](#) completely gets rid of recurrence and only uses attention to model the dependencies among words. For example, we can use attention to incorporate the representations from  $y_1$  to  $y_t$  when predicting  $y_{t+1}$ , simply by attending to their word embeddings. This is called *decoder self-attention*.

**Question:** By getting rid of recurrence and only using decoder self-attention, can we compute the logits for any two different words  $y_{t_1}$  and  $y_{t_2}$  in parallel at training time (only consider decoder for now)? Why?

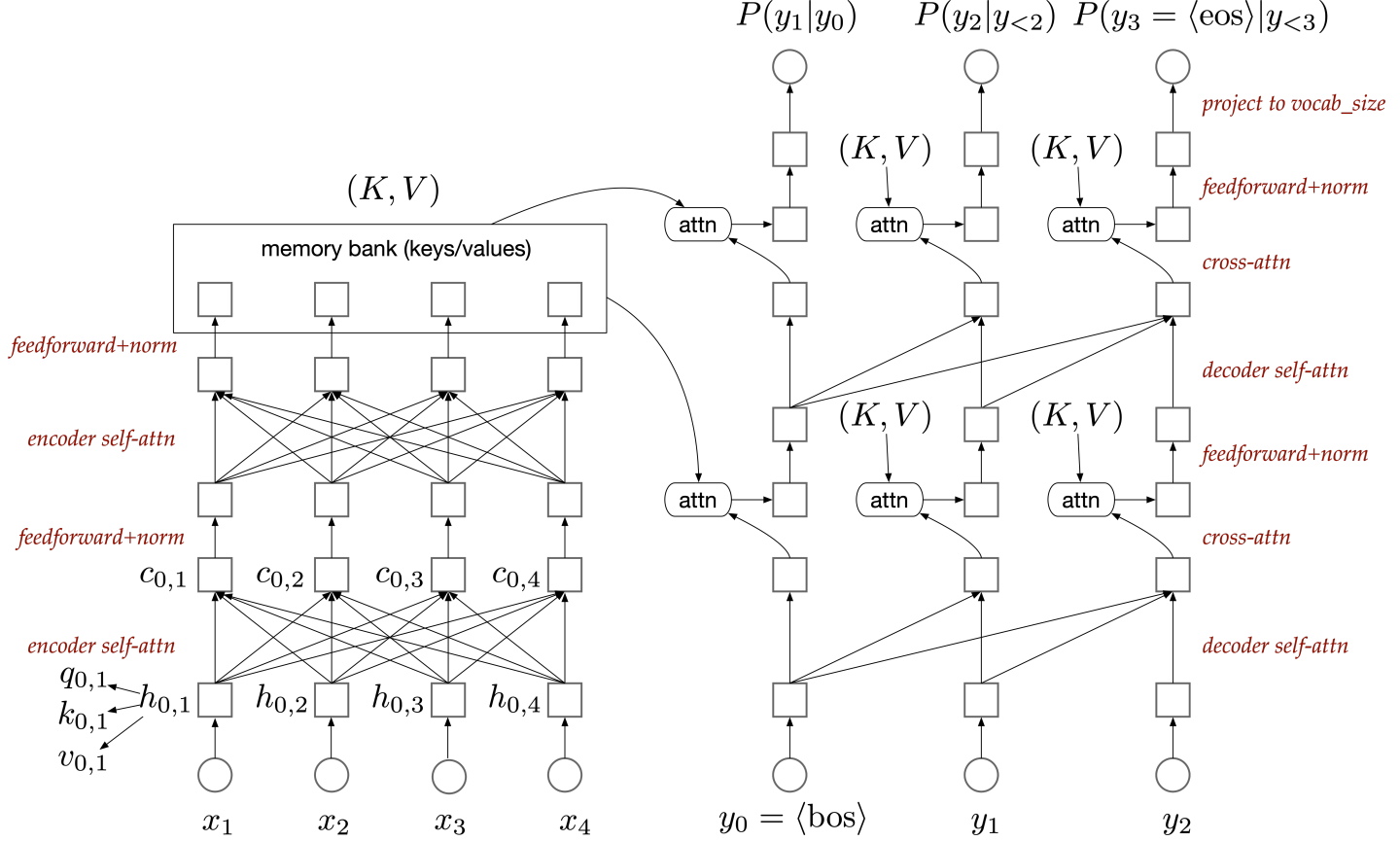
**In general, we can not compute the logits for any two different words in parallel, since there is a dependency between previous logits to future logits, and that apply to training time as well to inference.**

**However, According to papers online and few notes in our reading material, in practice the decoding using self-attention during training is done in parallel, mostly thanks to a method called 'Teacher Forcing'. The idea behind this method is that instead of taking the previous output of the model at step  $t$  as input at step  $t+1$ , we can use the ground truth output at step  $t$  (that's like using our output as part of the input). That works since during training, the output words predicted previously are known, because they are taken from the target side (ground truth) of our training data (which is parallel). This way, we avoid the dependency between logits of different time steps.**

Similarly, at the encoder side, for each word  $x_i$ , we let it attend to the embeddings of  $x_1, \dots, x_S$ , to model the context in which  $x_i$  appears. This is called *encoder self-attention*. It is different from decoder self-attention in that here every word attends to all words, but at the decoder side, every word can only attend to the previous words (since the prediction of word  $y_t$  cannot use the information from any  $y_{\geq t}$ ).

To incorporate source-side information at the decoder side, at each time step, we let the decoder attend to the top-layer encoder outputs, as we did in the RNN-based encoder-decoder model above. This is called *cross-attention*. Note that there's no initialization of decoder hidden state here, since we no longer use an RNN.

The process we describe above is only a single layer of attention. In practice, transformers stack multiple layers of attention and feedforward layers, using the outputs from the layer below as the inputs to the layer above, as shown in the illustration below.



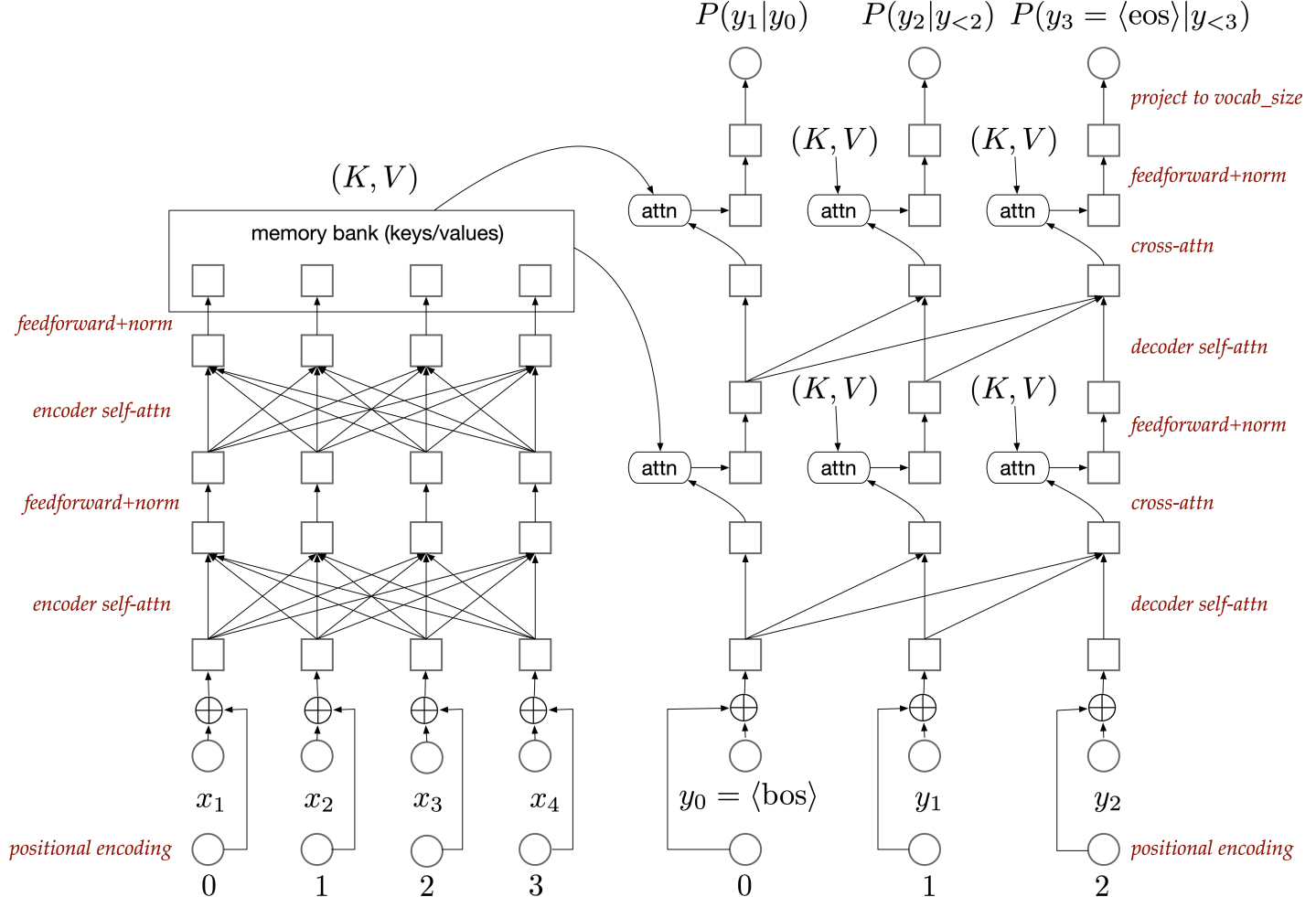
In the above illustration, due to space limits, we omitted the details of encoder self-attention and decoder self-attention, and we describe it here, using encoder-self-attention at layer 0 as an example. First, we use three linear projections to project each hidden state  $h_{0,i}$  to a query vector  $q_{0,i}$ , a key vector  $k_{0,i}$ , and a value vector  $v_{0,i}$ . Then at each position  $i$ , we use  $q_i$  as the query, and  $\{(k_{0,j}, v_{0,j}) : j \in \{1, \dots, S\}\}$  as keys/values to produce a context vector  $c_{0,i}$ . Note that the keys/values are the same for different positions, and the only difference is that a different query vector is used for each position.

A clear difference between the transformer architecture and the RNN-based encoder decoder architecture is that there are no horizontal arrows in the transformer model: transformers only use position-wise operations and attention operations. The dependencies among words are **only introduced by the attention operations**, while the other operations such as feedforward, nonlinearity, and normalization are position-wise, that is, they do not depend on other positions, and can thus be performed in parallel.

**Question:** In the above transformer model, if we shuffle the input words  $x_1, \dots, x_4$ , would we get a different distribution over  $y$ ? Why or why not?

**No, we would get the same distribution over  $y$ , since we don't use positional embedding (As described later and described in our reading material).**

Since the transformer model itself doesn't have any sense of position/order, we encode the position of the word in the sentence, and add it to the word embedding as the input representation, as illustrated below.



The illustrations above also omitted residual connections, which add the inputs to certain operations (such as attention and feedforward) to the outputs. More details can be found in the code below.

## ▼ Causal attention mask

To efficiently train the transformer model, we want to batch the attention operations together such that they can be fully parallelized along the sequence length dimension. (The non-attention operations are position-wise so they are trivially parallelizable.) This is quite straightforward for encoder self-attention and decoder-encoder cross-attention given our batched implementation of the `attention` function. However, things are a bit trickier for the decoder: each word  $y_t$  attends to  $t - 1$  previous words  $y_1, \dots, y_{t-1}$ , which means each word  $y_t$  has a different set of key-value pairs. Is it possible to batch them together?

The solution is to use *attention masks*. For every word  $y_t$ , we give it all key-value pairs at  $y_1, \dots, y_T$ , and we disallow attending to future words  $y_t, y_{t+1}, \dots, y_T$  through an attention mask. (Recall that the `attention` function takes a `mask` argument.) We usually call this attention mask a *causal attention mask*, as it prevents the leakage of information from the future into the past. Since every  $y_t$  has the same set of (key, value) pairs, we can batch them and compute the context vectors using a single call to the function `attention`.

What should such a mask be? Implement the `causal_mask` function below to generate this mask.

Hint: you might find [torch.triu](#) useful.

```
1 #TODO - implement this function, which returns a causal attention mask
2 def causal_mask(T):
3     """
4     Generate a causal mask.
5     Arguments:
6         T: the length of target sequence
7     Returns:
8         mask: a T x T tensor, where `mask[i, j]` should be `True`
9             if  $y_i$  can attend to  $y_{j-1}$  (there's a "-1" since the first
10            token in decoder input is  $\langle \text{bos} \rangle$ ) and `False` if  $y_i$  cannot
11            attend to  $y_{j-1}$ 
12     """
13     mask = torch.ones((T,T), dtype=torch.bool)
14     mask = torch.triu(mask)
15     mask = torch.transpose(mask, 0, 1)
16     return mask.to(device)
```

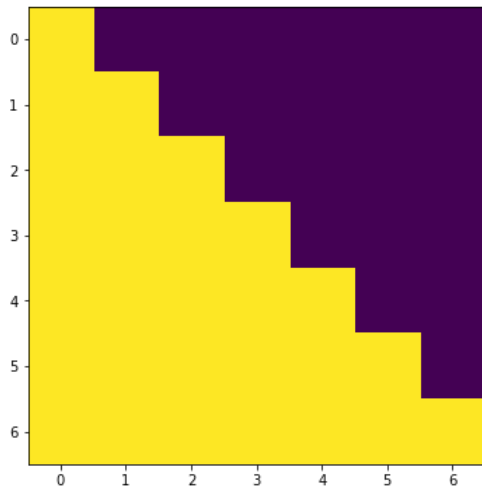
```
1 grader.check("causal_attention_mask")
```

All tests passed!

We can visualize the attention mask and manually check if it's what we expected.

```
1 fig, ax = plt.subplots(figsize=(8, 6))
2
3 T = 7
4 mask = causal_mask(T)
5 ax.imshow(mask.cpu())
6
7 # Uncomment the line below if the plot does not show up
8 # Make sure to comment that before submitting to gradescope
9 # since there would be some autograder issues with `plt.show()`
10 #plt.show()
```

<matplotlib.image.AxesImage at 0x7f3fb4740ed0>



As we have emphasized multiple times, unlike RNN-based encoder-decoders, transformer encoder/decoders are parallelizable in the sequence length dimension, even for the decoder: by using causal masks, all positions (at the same layer) can be computed all at once (if the lower layer has been computed). The parallelizability of transformers is the key to its success since it allows for training it on vast amounts of data.

Now we are ready to complete the implementation of the transformer model. The code is structured as a set of classes:

`TransformerEncoderLayer`\*, `TransformerEncoder`, `TransformDecoderLayer`\*, `TransformDecoder`, `PositionalEmbedding`, and `TransformerEncoderDecoder`\*. We've provided almost all the necessary code. In particular, we provide code for all position-wise operations.

Your job is only to implement the parts involving attention and to figure out the correct attention masks, which involves only the three classes marked above with a star.

Hint: Completing this transformer implementation should require very little code, just a few lines.

Hint: The causal mask is a 2-D matrix, but we want to add a batch dimension, and expand it to be of the desired size. For this purpose, you can use [torch.repeat](#).

```
1 #TODO - implement `forward_encoder` and `forward_decoder`.
2 # `TransformerEncoderDecoder` inherits most functions from `AttnEncoderDecoder`
3 class TransformerEncoderDecoder(AttnEncoderDecoder):
4     def __init__(self, src_field, tgt_field, hidden_size=64, layers=3):
5         """
6         Initializer. Creates network modules and loss function.
7         Arguments:
8             src_field: src field
9             tgt_field: tgt field
10            hidden_size: hidden layer size of both encoder and decoder
11            layers: number of layers of both encoder and decoder
12        """
13        super(AttnEncoderDecoder, self).__init__()
14        self.src_field = src_field
15        self.tgt_field = tgt_field
16
17        # Keep the vocabulary sizes available
18        self.V_src = len(src_field.vocab.itos)
19        self.V_tgt = len(tgt_field.vocab.itos)
20
21        # Get special word ids
22        self.padding_id_src = src_field.vocab.stoi[src_field.pad_token]
23        self.padding_id_tgt = tgt_field.vocab.stoi[tgt_field.pad_token]
24        self.bos_id = tgt_field.vocab.stoi[tgt_field.init_token]
25        self.eos_id = tgt_field.vocab.stoi[tgt_field.eos_token]
```



```

26 # Keep hyper-parameters available
27 self.embedding_size = hidden_size
28 self.hidden_size = hidden_size
29 self.layers = layers
30
31
32 # Create essential modules
33 self.encoder = TransformerEncoder(self.V_src, hidden_size, layers)
34 self.decoder = TransformerDecoder(self.V_tgt, hidden_size, layers)
35
36 # Final projection layer
37 self.hidden2output = nn.Linear(hidden_size, self.V_tgt)
38
39 # Create loss function
40 self.loss_function = nn.CrossEntropyLoss(reduction='sum',
41                                           ignore_index=self.padding_id_tgt)
42
43 def forward_encoder(self, src, src_lengths):
44     """
45     Encodes source words `src`.
46     Arguments:
47         src: src batch of size (max_src_len, bsz)
48         src_lengths: src lengths (bsz)
49     Returns:
50         memory_bank: a tensor of size (src_len, bsz, hidden_size)
51     """
52     # The reason we don't directly pass in src_mask as in `forward_decoder` is to
53     # enable us to reuse beam search implemented for RNN-based encoder-decoder
54     src_len = src.size(0)
55     #TODO - compute `encoder_self_attn_mask`
56     src_mask = src.ne(self.padding_id_src)
57     encoder_self_attn_mask = src_mask.transpose(0,1).unsqueeze(1).repeat(1,src_len,1)
58
59     memory_bank = self.encoder(src, encoder_self_attn_mask)
60     return memory_bank, None
61
62 def forward_decoder(self, tgt_in, memory_bank, src_mask):
63     """
64     Decodes based on memory bank, and ground truth target words.
65     Arguments:
66         tgt_in: a tensor of size (tgt_len, bsz)
67         memory_bank: a tensor of size (src_len, bsz, hidden_size), encoder outputs
68                     at every position
69         src_mask: a tensor of size (src_len, bsz) which is `False` for source paddings
70     Returns:
71         Logits of size (tgt_len, bsz, V_tgt) (before the softmax operation)
72     """
73     tgt_len = tgt_in.size(0)
74     bsz = tgt_in.size(1)
75     #TODO - compute `cross_attn_mask` and `decoder_self_attn_mask`
76
77     #0.75
78     cross_attn_mask = src_mask.repeat(tgt_len,1,1)
79     cross_attn_mask = cross_attn_mask.transpose(1,2)
80     cross_attn_mask = cross_attn_mask.transpose(0,1)
81     decoder_self_attn_mask = causal_mask(tgt_len).repeat(bsz,1,1)
82
83     outputs = self.decoder(tgt_in, memory_bank, cross_attn_mask, decoder_self_attn_mask)
84     logits = self.hidden2output(outputs)
85     return logits
86
87 def forward(self, src, src_lengths, tgt_in):
88     """
89     Performs forward computation, returns logits.
90     Arguments:
91         src: src batch of size (max_src_len, bsz)
92         src_lengths: src lengths of size (bsz)
93         tgt_in: a tensor of size (tgt_len, bsz)
94     """
95     src_mask = src.ne(self.padding_id_src) # max_src_len, bsz
96     # Forward encoder
97     memory_bank, _ = self.forward_encoder(src, src_lengths)
98     # Forward decoder
99     logits = self.forward_decoder(tgt_in, memory_bank, src_mask)
100     return logits
101
102 def forward_decoder_incrementally(self, prev_decoder_states, tgt_in_onestep,
103                                   memory_bank, src_mask, normalize=True):
104     """
105     Forward the decoder at `decoder_state` for a single step with token `tgt_in_onestep`.
106     This function will be used in beam search. Note that the implementation here is
107     very inefficient, since we do not cache any decoder state, but instead we only
108     cache previously generated tokens in `prev_decoder_states`, and do a fresh
109     `forward_decoder`.

```

```

Arguments:
111     prev_decoder_states: previous tgt words. None for the first step.
112     tgt_in_onestep: a tensor of size (bsz), tokens at one step
113     memory_bank: a tensor of size (src_len, bsz, hidden_size), src hidden states
114                   at every position
115     src_mask: a tensor of size (src_len, bsz): a boolean tensor, `False` where
116               src is padding.
117     normalize: use log_softmax to normalize or not. Beam search needs to normalize,
118               while `forward_decoder` does not
119 Returns:
120     logits: Log probabilities for `tgt_in_token` of size (bsz, V_tgt)
121     decoder_states: we use tgt words up to now as states, a tensor of size (len, bsz)
122     None: to keep output format the same as AttnEncoderDecoder, such that we can
123           reuse beam search code
124
125     """
126     prev_tgt_in = prev_decoder_states # tgt_len, bsz
127     src_len = memory_bank.size(0)
128     bsz = memory_bank.size(1)
129     tgt_in_onestep = tgt_in_onestep.view(1, -1) # 1, bsz
130     if prev_tgt_in is not None:
131         tgt_in = torch.cat((prev_tgt_in, tgt_in_onestep), 0) # tgt_len+1, bsz
132     else:
133         tgt_in = tgt_in_onestep
134     tgt_len = tgt_in.size(1)
135
136     logits = self.forward_decoder(tgt_in, memory_bank, src_mask)
137     logits = logits[-1]
138     if normalize:
139         logits = torch.log_softmax(logits, dim=-1)
140     decoder_states = tgt_in
141     return logits, decoder_states, None

```

```

1 class TransformerEncoder(nn.Module):
2     r"""TransformerEncoder is an embedding layer and a stack of N encoder layers.
3     Arguments:
4         hidden_size: hidden size.
5         layers: the number of encoder layers.
6     """
7
8     def __init__(self, vocab_size, hidden_size, layers):
9         super().__init__()
10        self.embed = PositionalEmbedding(vocab_size, hidden_size)
11        self.encoder_layer = TransformerEncoderLayer(hidden_size)
12        self.layers = _get_clones(self.encoder_layer, layers)
13        self.norm = nn.LayerNorm(hidden_size)
14
15    def forward(self, src, encoder_self_attn_mask):
16        r"""Pass the input through the word embedding layer, followed by
17        the encoder layers in turn.
18        Arguments:
19            src: src batch of size (max_src_len, bsz)
20            encoder_self_attn_mask: the mask for encoder self-attention, it's of size
21                                   (bsz, max_src_len, max_src_len)
22        Returns:
23            a tensor of size (max_src_len, bsz, hidden_size)
24        """
25        output = self.embed(src)
26        for mod in self.layers:
27            output = mod(output, encoder_self_attn_mask=encoder_self_attn_mask)
28        output = self.norm(output)
29        return output
30
31
32 class TransformerEncoderLayer(nn.Module):
33     r"""TransformerEncoderLayer is made up of self-attn and feedforward network.
34     Arguments:
35         hidden_size: hidden size.
36     """
37
38     def __init__(self, hidden_size):
39         super(TransformerEncoderLayer, self).__init__()
40         self.hidden_size = hidden_size
41         fwd_hidden_size = hidden_size * 4
42
43         # Create modules
44         self.linear1 = nn.Linear(hidden_size, fwd_hidden_size)
45         self.linear2 = nn.Linear(fwd_hidden_size, hidden_size)
46         self.norm1 = nn.LayerNorm(hidden_size)
47         self.norm2 = nn.LayerNorm(hidden_size)
48         self.activation = nn.ReLU()
49         # Attention related
50         self.q_proj = nn.Linear(hidden_size, hidden_size)
51         self.k_proj = nn.Linear(hidden_size, hidden_size)

```

```

52 self.v_proj = nn.Linear(hidden_size, hidden_size)
53 self.context_proj = nn.Linear(hidden_size, hidden_size)
54
55
56 def forward(self, src, encoder_self_attn_mask):
57     r"""Pass the input through the encoder layer.
58     Arguments:
59         src: an input tensor of size (max_src_len, bsz, hidden_size).
60         encoder_self_attn_mask: attention mask of size (bsz, max_src_len, max_src_len),
61             it's `False` where the corresponding attention is disabled
62     Returns:
63         a tensor of size (max_src_len, bsz, hidden_size).
64     """
65     # Attend
66     q = self.q_proj(src) / math.sqrt(self.hidden_size) # a trick needed to make transformer work
67     k = self.k_proj(src)
68     v = self.v_proj(src)
69     #TODO - compute `context`
70     _, context = attention(q, k, v, encoder_self_attn_mask) # ours
71     src2 = self.context_proj(context) # ours
72     # Residual connection
73     src = src + src2
74     src = self.norm1(src)
75     # Feedforward for each position
76     src2 = self.linear2(self.activation(self.linear1(src)))
77     src = src + src2
78     src = self.norm2(src)
79     return src
80
81
82 class TransformerDecoder(nn.Module):
83     r"""TransformerDecoder is an embedding layer and a stack of N decoder layers.
84     Arguments:
85         hidden_size: hidden size.
86         layers: the number of sub-encoder-layers in the encoder.
87     """
88     def __init__(self, vocab_size, hidden_size, layers):
89         super(TransformerDecoder, self).__init__()
90         self.embed = PositionalEmbedding(vocab_size, hidden_size)
91         decoder_layer = TransformerDecoderLayer(hidden_size)
92         self.layers = _get_clones(decoder_layer, layers)
93         self.norm = nn.LayerNorm(hidden_size)
94
95     def forward(self, tgt_in, memory, cross_attn_mask, decoder_self_attn_mask):
96         r"""Pass the inputs (and mask) through the word embedding layer, followed by
97         the decoder layer in turn.
98     Arguments:
99         tgt_in: tgt batch of size (max_tgt_len, bsz)
100         memory: the outputs of the encoder (max_src_len, bsz, hidden_size)
101         cross_attn_mask: attention mask of size (bsz, max_tgt_len, max_src_len),
102             it's `False` where the cross-attention is disallowed.
103         decoder_self_attn_mask: attention mask of size (bsz, max_tgt_len, max_tgt_len),
104             it's `False` where the self-attention is disallowed.
105     Returns:
106         a tensor of size (max_tgt_len, bsz, hidden_size)
107     """
108     output = self.embed(tgt_in)
109     for mod in self.layers:
110         output = mod(output, memory, cross_attn_mask=cross_attn_mask, \
111                     decoder_self_attn_mask=decoder_self_attn_mask)
112
113     output = self.norm(output)
114     return output
115
116
117 class TransformerDecoderLayer(nn.Module):
118     r"""TransformerDecoderLayer is made up of self-attn, cross-attn, and
119     feedforward network.
120     Arguments:
121         hidden_size: hidden size.
122     """
123
124     def __init__(self, hidden_size):
125         super(TransformerDecoderLayer, self).__init__()
126         self.hidden_size = hidden_size
127         fwd_hidden_size = hidden_size * 4
128
129         # Create modules
130         self.linear1 = nn.Linear(hidden_size, fwd_hidden_size)
131         self.linear2 = nn.Linear(fwd_hidden_size, hidden_size)
132
133         self.activation = nn.ReLU()
134
135         self.norm1 = nn.LayerNorm(hidden_size)

```

```

136 self.norm2 = nn.LayerNorm(hidden_size)
137 self.norm3 = nn.LayerNorm(hidden_size)
138
139 # Attention related
140 self.q_proj_self = nn.Linear(hidden_size, hidden_size)
141 self.k_proj_self = nn.Linear(hidden_size, hidden_size)
142 self.v_proj_self = nn.Linear(hidden_size, hidden_size)
143 self.context_proj_self = nn.Linear(hidden_size, hidden_size)
144
145 self.q_proj_cross = nn.Linear(hidden_size, hidden_size)
146 self.k_proj_cross = nn.Linear(hidden_size, hidden_size)
147 self.v_proj_cross = nn.Linear(hidden_size, hidden_size)
148 self.context_proj_cross = nn.Linear(hidden_size, hidden_size)
149
150 def forward(self, tgt, memory, cross_attn_mask, decoder_self_attn_mask):
151     r"""Pass the inputs (and mask) through the decoder layer.
152     Arguments:
153         tgt: an input tensor of size (max_tgt_len, bsz, hidden_size).
154         memory: encoder outputs of size (max_src_len, bsz, hidden_size).
155         cross_attn_mask: attention mask of size (bsz, max_tgt_len, max_src_len),
156             it's `False` where the cross-attention is disallowed.
157         decoder_self_attn_mask: attention mask of size (bsz, max_tgt_len, max_tgt_len),
158             it's `False` where the self-attention is disallowed.
159     Returns:
160         a tensor of size (max_tgt_len, bsz, hidden_size)
161     """
162     # Self attention (decoder-side)
163     q = self.q_proj_self(tgt) / math.sqrt(self.hidden_size)
164     k = self.k_proj_self(tgt)
165     v = self.v_proj_self(tgt)
166     #TODO - compute `context`
167     _, context = attention(q, k, v, decoder_self_attn_mask) # ours
168     tgt2 = self.context_proj_self(context) # ours
169     tgt = tgt + tgt2
170     tgt = self.norm1(tgt)
171     # Cross attention (decoder attends to encoder)
172     q = self.q_proj_cross(tgt) / math.sqrt(self.hidden_size)
173     k = self.k_proj_cross(memory)
174     v = self.v_proj_cross(memory)
175     #TODO - compute `context`
176     _, context = attention(q, k, v, cross_attn_mask) # ours
177     tgt2 = self.context_proj_cross(context)
178     tgt = tgt + tgt2
179     tgt = self.norm2(tgt)
180     tgt2 = self.linear2(self.activation(self.linear1(tgt)))
181     tgt = tgt + tgt2
182     tgt = self.norm3(tgt)
183     return tgt
184
185 class PositionalEmbedding(nn.Module):
186     """Embeds a word both by its word id and by its position in the sentence."""
187     def __init__(self, vocab_size, embedding_size, max_len=1024):
188         super(PositionalEmbedding, self).__init__()
189         self.embedding_size = embedding_size
190
191         self.embed = nn.Embedding(vocab_size, embedding_size)
192         pe = torch.zeros(max_len, embedding_size)
193         position = torch.arange(0, max_len).unsqueeze(1)
194         div_term = torch.exp(torch.arange(0, embedding_size, 2) *
195                               -(math.log(10000.0) / embedding_size))
196         pe[:, 0::2] = torch.sin(position * div_term)
197         pe[:, 1::2] = torch.cos(position * div_term)
198         pe = pe.unsqueeze(1) # max_len, 1, embedding_size
199         self.register_buffer('pe', pe)
200
201     def forward(self, batch):
202         x = self.embed(batch) * math.sqrt(self.embedding_size) # type embedding
203         # Add positional encoding to type embedding
204         x = x + self.pe[:x.size(0)].detach()
205         return x
206
207
208 def _get_clones(module, N):
209     """Copies a module `N` times"""
210     return nn.ModuleList([copy.deepcopy(module) for i in range(N)])

```

```

1 EPOCHS = 2 # epochs, we highly recommend starting with a smaller number like 1
2 LEARNING_RATE = 2e-3 # learning rate
3
4 # Instantiate and train classifier
5 model_transformer = TransformerEncoderDecoder(SRC, TGT,
6         hidden_size     = 64,
7         layers           = 3,

```

```

8 ).to(device)
9
10 model_transformer.train_all(train_iter, val_iter, epochs=EPOCHS, learning_rate=LEARNING_RATE)
11 model_transformer.load_state_dict(model_transformer.best_model)

100%|██████████| 2032/2032 [00:53<00:00, 38.30it/s]
Epoch: 0 Training Perplexity: 2.2093 Validation Perplexity: 1.3195
100%|██████████| 2032/2032 [00:53<00:00, 37.80it/s]
Epoch: 1 Training Perplexity: 1.2673 Validation Perplexity: 1.1367
<All keys matched successfully>

```

You might notice that in these experiments training transformers doesn't appear to be faster than training RNNs. There are two reasons for that: first, we are not using GPUs; second, even if you use GPUs, the sequences here are too short to observe the benefits of parallelizing along the horizontal direction. In real datasets with long sentences, training transformers is much faster than training RNNs, so under the same computational budget, using transformers allows for training on much larger datasets. This is one of the primary reasons transformers dominate NLP research these days.

**Question:** Would there be any speed advantage of decoding (generation) using transformers compared to RNNs? Why or why not?

**No, there wouldn't be. As we explained above, there are dependencies between the generated outputs at different time steps. Hence, the decoding suffer the same problem of the recurrence in RNNs. As we explained above, in training time we can accelerate and parallel it using teaching-forcing, but in inference time we can not, and there wouldn't be speed advantage of decoding using transformer over RNNs.**

```

1 # Evaluate model performance, the expected value should be < 1.5
2 print (f'Test perplexity: {model_transformer.evaluate_ppl(test_iter):.3f}')

Test perplexity: 1.145

```

```

1 grader.check("transformer_ppl")

All tests passed!

```

Now that we have a trained model, we can decode from it using our previously implemented beam search function. If the code below throws any errors, you might need to modify your beam search code such that it generalizes here.

```

1 grader.check("transformer_beam_search")

All tests passed!

```

```

1 DEBUG_FIRST = 10 # set to False to disable printing predictions
2 K = 1 # beam size 1
3
4 correct = 0
5 total = 0
6
7 # create beam searcher
8 beam_searcher = BeamSearcher(model_transformer)
9
10 for index, batch in enumerate(test_iter, start=1):
11     # Input and output
12     src, src_lengths = batch.src
13     # Predict
14     model.all_attns = []
15     prediction, _ = beam_searcher.beam_search(src, src_lengths, K)
16     # Convert to string
17     prediction = ' '.join([TGT.vocab.itos[token] for token in prediction])
18     prediction = prediction.lstrip('<bos>').rstrip('<eos>').strip()
19     ground_truth = ' '.join([TGT.vocab.itos[token] for token in batch.tgt.view(-1)])
20     ground_truth = ground_truth.lstrip('<bos>').rstrip('<eos>').strip()
21     if DEBUG_FIRST > index:
22         src = ' '.join([SRC.vocab.itos[item] for item in src.view(-1)])
23         print (f'Source: {src}')
24         print (f'Prediction: {prediction}')
25         print (f'Ground truth: {ground_truth}')
26     if ground_truth == prediction:
27         correct += 1
28     total += 1
29
30 print (f'Accuracy: {correct/total:.2f}')

```

```

Source: sixteen thousand eight hundred and thirty two
Prediction:  1 6 8 3 2

```

Ground truth: 1 6 8 3 2  
Source: sixty seven million six hundred and eighty five thousand two hundred and thirty  
Prediction: 6 7 6 8 0 5 2 3 0  
Ground truth: 6 7 6 8 5 2 3 0  
Source: six thousand two hundred and twelve  
Prediction: 6 2 1 2  
Ground truth: 6 2 1 2  
Source: seven hundred and ninety eight million three hundred and thirty one thousand eight hundred and eighteen  
Prediction: 7 9 8 3 3 1 8 1 8  
Ground truth: 7 9 8 3 3 1 8 1 8  
Source: eighty eight million four hundred and thirteen thousand nine hundred and eighteen  
Prediction: 8 8 4 1 3 9 1 8  
Ground truth: 8 8 4 1 3 9 1 8  
Source: three hundred and seventy four thousand two hundred and seventy  
Prediction: 3 7 4 2 7 0  
Ground truth: 3 7 4 2 7 0  
Source: ninety eight million three hundred and seventy thousand five hundred and forty five  
Prediction: 9 8 3 7 0 5 4 5  
Ground truth: 9 8 3 7 0 5 4 5  
Source: ninety seven thousand seven hundred and sixty two  
Prediction: 9 7 7 6 2  
Ground truth: 9 7 7 6 2  
Source: four hundred and ten thousand two hundred and three  
Prediction: 4 1 0 2 0 3  
Ground truth: 4 1 0 2 0 3  
Accuracy: 0.73

**Question:** When we first introduced attention above, adding it to an RNN model, we noted that

The attention scores  $\mathbf{a}$  lie on a *simplex* (meaning  $a_i \geq 0$  and  $\sum_i a_i = 1$ ), which lends it some interpretability: the closer  $a_i$  is to 1, the more "relevant" a key  $k_i$  (and hence its value  $v_i$ ) is to the given query. We will observe this later in the lab: When we are about to predict the target word "3",  $a_i$  is close to 1 for the source word  $x_i = \text{"three"}$ .

Can we interpret the attentions in a multi-layer transformer similarly? If so, what would you expect the attention scores to correspond to? If not, explain why.

**From our understanding, such interpretability when using multi-layer transformer is much more complex. From reading online, there are tools to visualize the interpretability of multi-layer transformers and from playing with we get the idea that the attention gasp more complex connections in deeper layers.**

You might have noticed that the transformer model underperforms the RNN-based encoder-decoder on this particular task. This might be due to several reasons:

- Transformers tend to be data hungry, sometimes requiring billions of words to train.
- The transformer formulation presented in this lab is not in its full form: for instance, instead of only doing attention once at each position for each layer, researchers usually use multiple attention operations in the hope of capturing different aspects of "relevance", which is called "multi-headed attention". For example, one attention head might be focusing on pronoun resolution, while the other might be looking for similar contexts before.
- Transformers are usually sensitive to hyper-parameters and require heavy tuning. For example, while we used a fixed learning rate, researchers usually use a customized learning rate scheduler which first warms up the learning rate, and then gradually decreases it. If you are interested, more details can be found in [the original paper](#).

We also recommend the excellent pedagogic blog posts: [The Illustrated Transformer](#) and [The Annotated Transformer](#).

In real-world applications, many state-of-the-art NLP approaches are based on transformers, such as the fake news generator used by [GROVER](#) that you've seen in the Embedded EthiCS class. For further readings if you are interested, we recommend [BERT](#) and [GPT-3](#).

## ▾ Lab debrief

**Question:** We're interested in any thoughts your group has about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment on might include the following:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?
- Are there additions or changes you think would make the lab better?

but you should comment on whatever aspects you found especially positive or negative.

***This lab was very tough and unclear. we think we did not have the tools to handel it in the time we had for it. maybe more time to work on it at class or more lectures could help.***

## End of Lab 4-5

To double-check your work, the cell below will rerun all of the autograder tests.

```
1 grader.check_all()
```

**attention:**

All tests passed!

**beam\_search:**

All tests passed!

**causal\_attention\_mask:**

All tests passed!

**encoder\_decoder\_ppl:**

All tests passed!

**transformer\_beam\_search:**

All tests passed!

**transformer\_ppl:**

All tests passed!

✓ 11s completed at 9:59 PM

