

Reference http://cs231n.stanford.edu/slides/2021/lecture_11.pdf

Ran by Sushant Gautam as a part of Assignemnt 5 for DL for VI course.

Sushant Gautam

MSISE Department of Electronics and Computer Engineering Institute of Engineering, Thapathali Campus

18 March 2022

```
1 # This mounts your Google Drive to the Colab VM.
2 from google.colab import drive
3 drive.mount('/content/drive')
4
5
6 #.path.of.cs231n.folder.in.google.drive
7 FOLDERNAME.='MS-DL-Assignemnt5'
8 assert.FOLDERNAME.is.not.None,."[!].Enter.the.foldername."
9
10 #.Now.that.we've.mounted.your.Drive,.this.ensures.that
11 #.the.Python.interpreter.of.the.Colab.VM.can.load
12 # python files from within it.
13 import sys
14 basefold = '/content/drive/My Drive/{}'.format(FOLDERNAME)
15 sys.path.append(basefold)
16
17 # This downloads the COCO dataset to your Drive
18 # if it doesn't already exist.
19 %cd /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
20 !bash get_datasets.sh
21 %cd /content/drive/My\ Drive/$FOLDERNAME

Mounted at /content/drive
/content/drive/My Drive/MS-DL-Assignemnt5/cs231n/datasets
/content/drive/My Drive/MS-DL-Assignemnt5
```

▼ Image Captioning with Transformers

You have now implemented a vanilla RNN and for the task of image captioning. In this notebook you will implement key pieces of a transformer decoder to accomplish the same task.

NOTE: This notebook will be primarily written in PyTorch rather than NumPy, unlike the RNN notebook.

```
1 # Setup cell.
2 import time, os, json
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
7 from cs231n.transformer_layers import *
8 from cs231n.captioning_solver_transformer import CaptioningSolverTransformer
9 from cs231n.classifiers.transformer import CaptioningTransformer
10 from cs231n.coco_utils import load_coco_data, sample_coco_minibatch, decode_captions
11 from cs231n.image_utils import image_from_url
12
13 %matplotlib inline
14 plt.rcParams['figure.figsize'] = (10.0, 8.0) # Set default size of plots.
15 plt.rcParams['image.interpolation'] = 'nearest'
16 plt.rcParams['image.cmap'] = 'gray'
17
18 %load_ext autoreload
19 %autoreload 2
20
21 def rel_error(x, y):
22     """ returns relative error """
23     return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

▼ COCO Dataset

As in the previous notebooks, we will use the COCO dataset for captioning.

```
1 # Load COCO data from disk into a dictionary.
2 data = load_coco_data(pca_features=True)
3
4 # Print out all the keys and values from the data dictionary.
5 for k, v in data.items():
6     if type(v) == np.ndarray:
7         print(k, type(v), v.shape, v.dtype)
8     else:
9         print(k, type(v), len(v))

base dir /content/drive/My Drive/MS-DL-Assignemnt5/cs231n/datasets/coco_captioning
/content/drive/My Drive/MS-DL-Assignemnt5/cs231n/datasets/coco_captioning/train2014_vgg16_fc7_pca.h5
train_captions <class 'numpy.ndarray'> (400135, 17) int32
train_image_idxs <class 'numpy.ndarray'> (400135,) int32
val_captions <class 'numpy.ndarray'> (195954, 17) int32
val_image_idxs <class 'numpy.ndarray'> (195954,) int32
train_features <class 'numpy.ndarray'> (82783, 512) float32
val_features <class 'numpy.ndarray'> (40504, 512) float32
idx_to_word <class 'list'> 1004
word_to_idx <class 'dict'> 1004
train_urls <class 'numpy.ndarray'> (82783,) <U63
val_urls <class 'numpy.ndarray'> (40504,) <U63
```

Transformer

As you have seen, RNNs are incredibly powerful but often slow to train. Further, RNNs struggle to encode long-range dependencies (though LSTMs are one way of mitigating the issue). In 2017, Vaswani et al introduced the Transformer in their paper ["Attention Is All You Need"](#) to a) introduce parallelism and b) allow models to learn long-range dependencies. The paper not only led to famous models like BERT and GPT in the natural language processing community, but also an explosion of interest across fields, including vision. While here we introduce the model in the context of image captioning, the idea of attention itself is much more general.

▼ Transformer: Multi-Headed Attention

Dot-Product Attention

Recall that attention can be viewed as an operation on a query $q \in \mathbb{R}^d$, a set of value vectors $\{v_1, \dots, v_n\}, v_i \in \mathbb{R}^d$, and a set of key vectors $\{k_1, \dots, k_n\}, k_i \in \mathbb{R}^d$, specified as

$$c = \sum_{i=1}^n v_i \alpha_i \alpha_i = \frac{\exp(k_i^\top q)}{\sum_{j=1}^n \exp(k_j^\top q)}$$

where α_i are frequently called the "attention weights", and the output $c \in \mathbb{R}^d$ is a correspondingly weighted average over the value vectors.

▼ Self-Attention

In Transformers, we perform self-attention, which means that the values, keys and query are derived from the input $X \in \mathbb{R}^{\ell \times d}$, where ℓ is our sequence length. Specifically, we learn parameter matrices $V, K, Q \in \mathbb{R}^{d \times d}$ to map our input X as follows:

$$\begin{aligned} v_i &= Vx_i \quad i \in \{1, \dots, \ell\} \\ k_i &= Kx_i \quad i \in \{1, \dots, \ell\} \\ q_i &= Qx_i \quad i \in \{1, \dots, \ell\} \end{aligned}$$

▼ Multi-Headed Scaled Dot-Product Attention

In the case of multi-headed attention, we learn a parameter matrix for each head, which gives the model more expressivity to attend to different parts of the input. Let h be number of heads, and Y_i be the attention output of head i . Thus we learn individual matrices Q_i, K_i and V_i . To keep our overall computation the same as the single-headed case, we choose $Q_i \in \mathbb{R}^{d \times d/h}, K_i \in \mathbb{R}^{d \times d/h}$ and $V_i \in \mathbb{R}^{d \times d/h}$. Adding in a scaling term $\frac{1}{\sqrt{d/h}}$ to our simple dot-product attention above, we have

$$Y_i = \text{softmax}\left(\frac{(XQ_i)(XK_i)^\top}{\sqrt{d/h}}\right)(XV_i)$$

where $Y_i \in \mathbb{R}^{\ell \times d/h}$, where ℓ is our sequence length.

In our implementation, we then apply dropout here (though in practice it could be used at any step):

$$Y_i = \text{dropout}(Y_i)$$

Finally, then the output of the self-attention is a linear transformation of the concatenation of the heads:

$$Y = [Y_1; \dots; Y_h]A$$

were $A \in \mathbb{R}^{d \times d}$ and $[Y_1; \dots; Y_h] \in \mathbb{R}^{\ell \times d}$.

Implement multi-headed scaled dot-product attention in the `MultiHeadAttention` class in the file `cs231n/transformer_layers.py`. The code below will check your implementation. The relative error should be less than 1e-3.

```
1 torch.manual_seed(231)
2
3 # Choose dimensions such that they are all unique for easier debugging:
4 # Specifically, the following values correspond to N=1, H=2, T=3, E//H=4, and E=8.
5 batch_size = 1
6 sequence_length = 3
7 embed_dim = 8
8 attn = MultiHeadAttention(embed_dim, num_heads=2)
9
10 # Self-attention.
11 data = torch.randn(batch_size, sequence_length, embed_dim)
12 self_attn_output = attn(query=data, key=data, value=data)
13
14 # Masked self-attention.
15 mask = torch.randn(sequence_length, sequence_length) < 0.5
16 masked_self_attn_output = attn(query=data, key=data, value=data, attn_mask=mask)
17
18 # Attention using two inputs.
19 other_data = torch.randn(batch_size, sequence_length, embed_dim)
20 attn_output = attn(query=data, key=other_data, value=other_data)
21
22 expected_self_attn_output = np.asarray([[
23 [-0.2494,  0.1396,  0.4323, -0.2411, -0.1547,  0.2329, -0.1936,
24      -0.1444],
25      [-0.1997,  0.1746,  0.7377, -0.3549, -0.2657,  0.2693, -0.2541,
26      -0.2476],
27      [-0.0625,  0.1503,  0.7572, -0.3974, -0.1681,  0.2168, -0.2478,
28      -0.3038]]])
29
30 expected_masked_self_attn_output = np.asarray([[
31 [-0.1347,  0.1934,  0.8628, -0.4903, -0.2614,  0.2798, -0.2586,
32      -0.3019],
33      [-0.1013,  0.3111,  0.5783, -0.3248, -0.3842,  0.1482, -0.3628,
34      -0.1496],
35      [-0.2071,  0.1669,  0.7097, -0.3152, -0.3136,  0.2520, -0.2774,
36      -0.2208]]])
37
38 expected_attn_output = np.asarray([[
39 [-0.1980,  0.4083,  0.1968, -0.3477,  0.0321,  0.4258, -0.8972,
40      -0.2744],
41      [-0.1603,  0.4155,  0.2295, -0.3485, -0.0341,  0.3929, -0.8248,
42      -0.2767],
43      [-0.0908,  0.4113,  0.3017, -0.3539, -0.1020,  0.3784, -0.7189,
44      -0.2912]]])
45
46 print('self_attn_output error: ', rel_error(expected_self_attn_output, self_attn_output.detach().numpy()))
47 print('masked_self_attn_output error: ', rel_error(expected_masked_self_attn_output, masked_self_attn_output.detach().numpy()))
48 print('attn_output error: ', rel_error(expected_attn_output, attn_output.detach().numpy()))

```

self_attn_output error: 0.0003775124598178026
masked_self_attn_output error: 0.0001526367643724865

attn_output error: 0.0003527921483788199

▼ Positional Encoding

While transformers are able to easily attend to any part of their input, the attention mechanism has no concept of token order. However, for many tasks (especially natural language processing), relative token order is very important. To recover this, the authors add a positional encoding to the embeddings of individual word tokens.

Let us define a matrix $P \in \mathbb{R}^{l \times d}$, where $P_{ij} =$

$$\begin{cases} \sin\left(i \cdot 10000^{-\frac{j}{d}}\right) & \text{if } j \text{ is even} \\ \cos\left(i \cdot 10000^{-\frac{(j-1)}{d}}\right) & \text{otherwise} \end{cases}$$

Rather than directly passing an input $X \in \mathbb{R}^{l \times d}$ to our network, we instead pass $X + P$.

Implement this layer in `PositionalEncoding` in `cs231n/transformer_layers.py`. Once you are done, run the following to perform a simple test of your implementation. You should see errors on the order of e^{-3} or less.

```
1 torch.manual_seed(231)
2
3 batch_size = 1
4 sequence_length = 2
5 embed_dim = 6
6 data = torch.randn(batch_size, sequence_length, embed_dim)
7
8 pos_encoder = PositionalEncoding(embed_dim)
9 output = pos_encoder(data)
10
11 expected_pe_output = np.asarray([[[-1.2340, 1.1127, 1.6978, -0.0865, -0.0000, 1.2728],
12                                   [ 0.9028, -0.4781, 0.5535, 0.8133, 1.2644, 1.7034]]]])
13
14 print('pe_output error: ', rel_error(expected_pe_output, output.detach().numpy()))

pe_output error: 0.00010421011374914356
```

Inline Question 1

Several key design decisions were made in designing the scaled dot product attention we introduced above. Explain why the following choices were beneficial:

1. Using multiple attention heads as opposed to one.
2. Dividing by $\sqrt{d/h}$ before applying the softmax function. Recall that d is the feature dimension and h is the number of heads.
3. Adding a linear transformation to the output of the attention operation. What would happen if we were to stack attention operations directly?

Only one or two sentences per choice is necessary, but be sure to be specific in addressing what would have happened without each given implementation detail, why such a situation would be suboptimal, and how the proposed implementation improves the situation.

Your Answer: Multiple attention heads allows for attending to parts of the sequence differently and helps model learn to catch longer-term dependencies and shorter-term dependencies with different attention strength.

The keys are not normalized. By virtue of the central limit theorem: the magnitude of the output from the dot product scales with the square root of the dimension of the keys. This makes the soft-max behave poorly. So dividing by the factor helps to normalize before applying softmax.

The operation of adding linear transformations just means to add their outputs, and the operation of scaling a linear transformation just means to scale its output.

▼ Overfit Transformer Captioning Model on Small Data

Run the following to overfit the Transformer-based captioning model on the same small dataset as we used for the RNN previously.

```
1 torch.manual_seed(231)
2 np.random.seed(231)
3
4 data = load_coco_data(max_train=50)
5
6 transformer = CaptioningTransformer(
7     word_to_idx=data['word_to_idx'],
8     input_dim=data['train_features'].shape[1],
9     wordvec_dim=256,
10    num_heads=2,
11    num_layers=2,
12    max_length=30
13)
14
15
16 transformer_solver = CaptioningSolverTransformer(transformer, data, idx_to_word=data['idx_to_word'],
17    num_epochs=100,
18    batch_size=25,
19    learning_rate=0.001,
20    verbose=True, print_every=10,
21)
22
23 transformer_solver.train()
24
25 # Plot the training losses.
26 plt.plot(transformer_solver.loss_history)
27 plt.xlabel('Iteration')
28 plt.ylabel('Loss')
29 plt.title('Training loss history')
30 plt.show()
```

Print final training loss. You should see a final loss of less than 0.03.

```
1 print('Final loss: ', transformer_solver.loss_history[-1])  
  
Final loss: 0.022183504
```

▼ Transformer Sampling at Test Time

The sampling code has been written for you. You can simply run the following to compare with the previous results with the RNN. As before the training results should be much better than the validation set results, given how little data we trained on.

```
1 # If you get an error, the URL just no longer exists, so don't worry!  
2 # You can re-sample as many times as you want.  
3 for split in ['train', 'val']:  
4     minibatch = sample_coco_minibatch(data, split=split, batch_size=2)  
5     gt_captions, features, urls = minibatch  
6     gt_captions = decode_captions(gt_captions, data['idx_to_word'])  
7  
8     sample_captions = transformer.sample(features, max_length=30)  
9     sample_captions = decode_captions(sample_captions, data['idx_to_word'])  
10  
11     for gt_caption, sample_caption, url in zip(gt_captions, sample_captions, urls):  
12         img = image_from_url(url)  
13         # Skip missing URLs.  
14         if img is None: continue  
15         plt.imshow(img)  
16         plt.title('%s\n%s\nGT:%s' % (split, sample_caption, gt_caption))  
17         plt.axis('off')  
18         plt.show()
```

▼ Caption Evalaution Using BLEU score

In addition to qualitatively evaluating your model by inspecting its results, you can also quantitatively evaluate your model using the BLEU unigram precision metric. In order to achieve full credit you should train a model that achieves a BLEU unigram score of >0.3. BLEU scores range from 0 to 1; the closer to 1, the better. Here's a reference to the [paper](#) that introduces BLEU if you're interested in learning more about how it works.

```
1 import nltk
2 def BLEU_score(gt_caption, sample_caption):
3     """
4     gt_caption: string, ground-truth caption
5     sample_caption: string, your model's predicted caption
6     Returns unigram BLEU score.
7     """
8     reference = [x for x in gt_caption.split(' ')
9                  if ('<END>' not in x and '<START>' not in x and '<UNK>' not in x)]
10    hypothesis = [x for x in sample_caption.split(' ')
11                 if ('<END>' not in x and '<START>' not in x and '<UNK>' not in x)]
12    BLEUScore = nltk.translate.bleu_score.sentence_bleu([reference], hypothesis, weights = [1])
13    return BLEUScore
14
15 def evaluate_model(model):
16     """
17     model: CaptioningRNN model
18     Prints unigram BLEU score averaged over 1000 training and val examples.
19     """
20     BLEUScores = {}
21     for split in ['train', 'val']:
22         minibatch = sample_coco_minibatch(data, split=split, batch_size=1000)
23         gt_captions, features, urls = minibatch
24         gt_captions = decode_captions(gt_captions, data['idx_to_word'])
25
26         sample_captions = model.sample(features)
27         sample_captions = decode_captions(sample_captions, data['idx_to_word'])
28
29
30         total_score = 0.0
31         for gt_caption, sample_caption, url in zip(gt_captions, sample_captions, urls):
32             total_score += BLEU_score(gt_caption, sample_caption)
33
34         BLEUScores[split] = total_score / len(sample_captions)
35
36     for split in BLEUScores:
37         print('Average BLEU score for %s: %f' % (split, BLEUScores[split]))
38
39
40
41 evaluate_model(transformer)
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
```

Average BLEU score for train: 0.992828
Average BLEU score for val: 0.213360