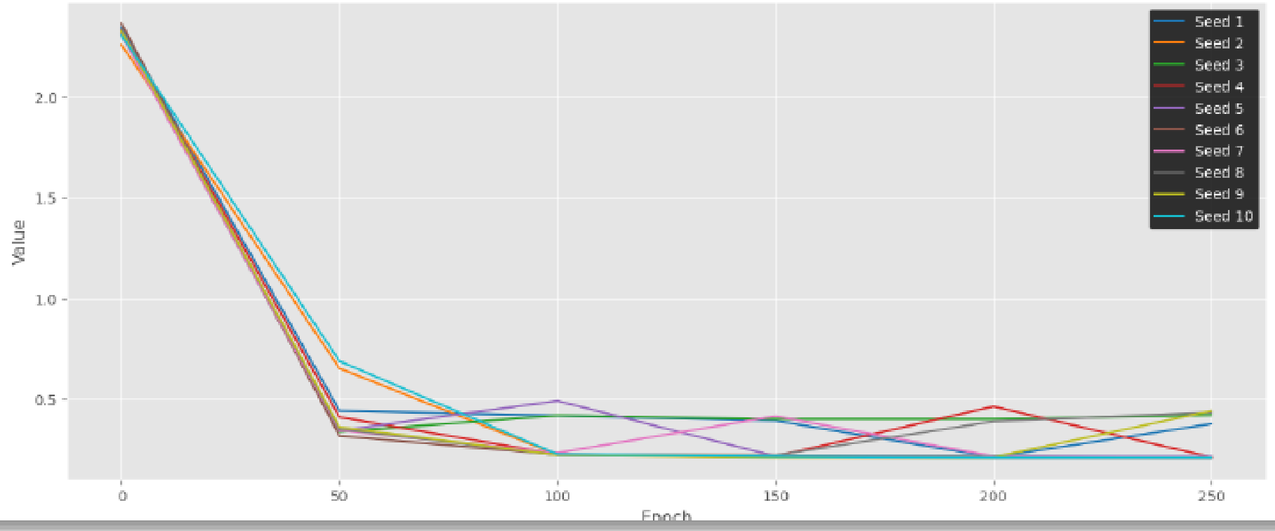
LET’S REVIEW OUR HOMEWORK

Excellent analysis

1. **Run a simple Artificial Neural Network (ANN) model 10 times using different random seeds. Use seeds ranging from 1 to 10 for these runs.**

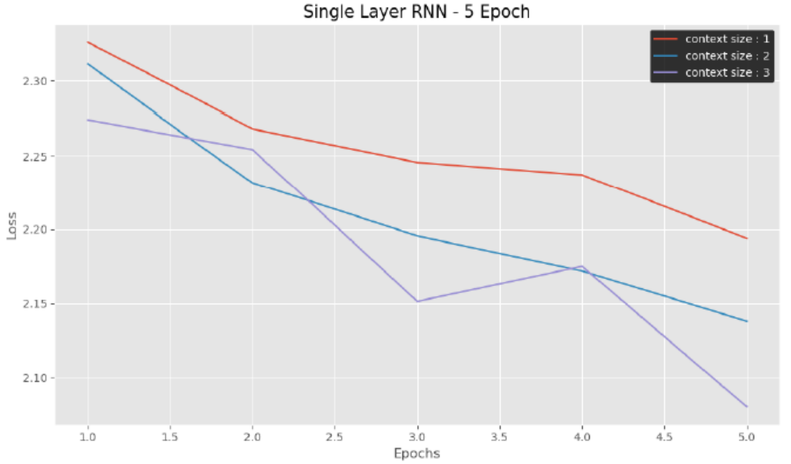
		Epoch					
		0	50	100	150	200	250
Seed	1	2.346	0.442	0.417	0.393	0.21	0.377
	2	2.259	0.652	0.223	0.218	0.215	0.209
	3	2.316	0.338	0.418	0.4	0.4	0.42
	4	2.329	0.41	0.225	0.215	0.463	0.21
	5	2.314	0.343	0.491	0.215	0.216	0.21
	6	2.365	0.317	0.223	0.214	0.21	0.21
	7	2.307	0.345	0.232	0.415	0.215	0.215
	8	2.336	0.351	0.221	0.22	0.389	0.433
	9	2.325	0.361	0.221	0.211	0.21	0.441
	10	2.305	0.688	0.225	0.215	0.21	0.21

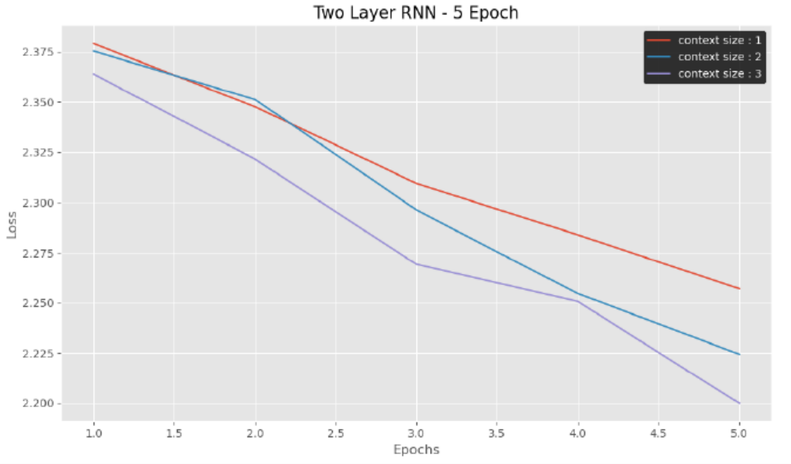


Excellent analysis

2. **Experiment: Train two types of Recurrent Neural Network (RNN) models—one with a single layer and the other with two layers. For each model, run experiments with varying numbers of training epochs (5, 50, 500) and different context sizes (1, 2, 3).**

		Single Layer RNN			Two Layer RNN		
		Epoch			Epoch		
		5	50	500	5	50	500
context sizes	1	2.194	1.07	0.238	2.257	0.938	0.237
	2	2.138	0.792	0.215	2.224	0.646	0.213
	3	2.08	0.771	0.216	2.2	0.633	0.216

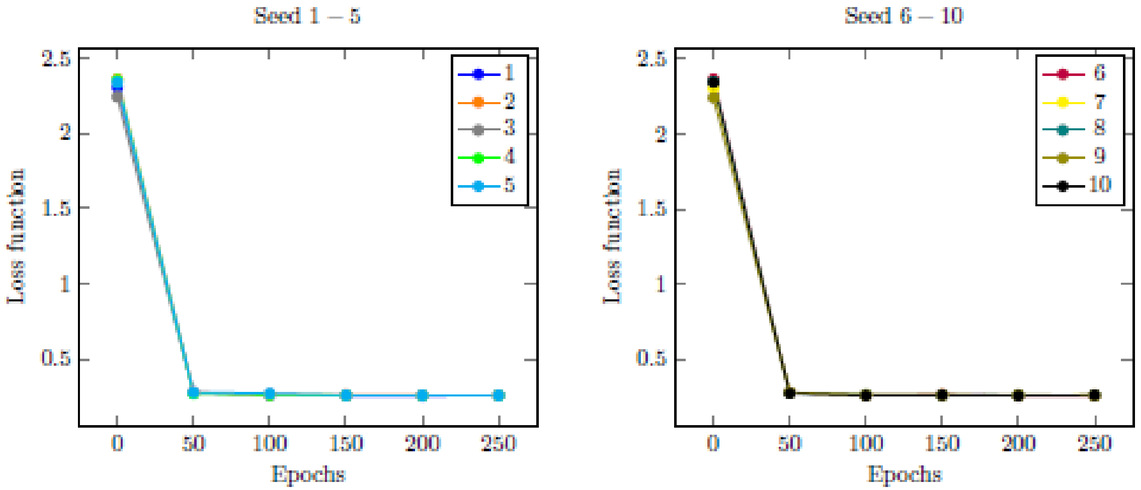




Abie Nugraha

And, here's the graphical representation for it.

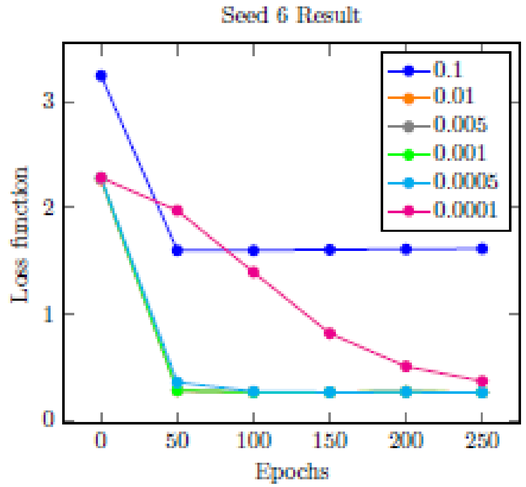
Figure 1: Using batch size of one



From here and left side of this table, it could be seen that all seeds are converging to similar value — around 0.25. The difference and variations within all experimented seeds are indistinguishable or negligible as well.

We could compare more, for example, by changing its learning rate. We will take seed 6's result to be experimented in different learning rate.

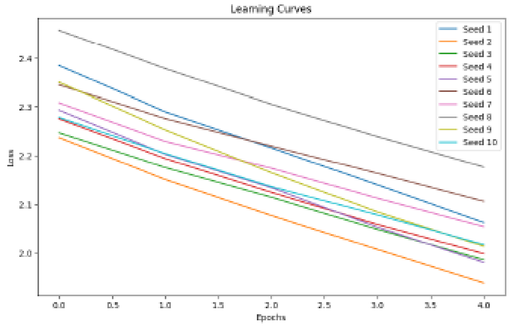
Figure 2: Different Learning Rate Comparison



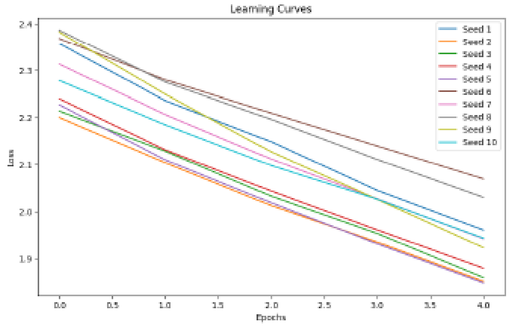
Efficient analysis

Layer of RNN	Context Sizes	Epoch	Final Loss value
1	1	5	2.0481736024220782
		50	0.4850320865710576
		500	0.2242969816705833
	2	5	1.9804903609412057
		50	0.4248789580804961
		500	0.2394988147508619
	3	5	2.012891641029945
		50	0.4445634661958768
		500	0.258040387255068
2	1	5	2.0907397190729777
		50	0.38734946548938753
		500	0.22330779161808703
	2	5	1.9624979070254736
		50	0.34522262641361784
		500	0.23876872490239162
	3	5	1.9331122636795044
		50	0.36903525401766485
		500	0.2560707236340162

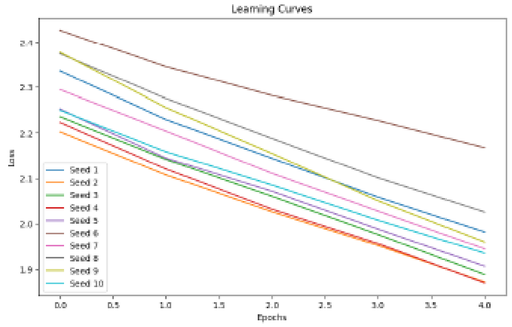
Table 4: Earliest Epoch for Stabilized/Converging Loss Values



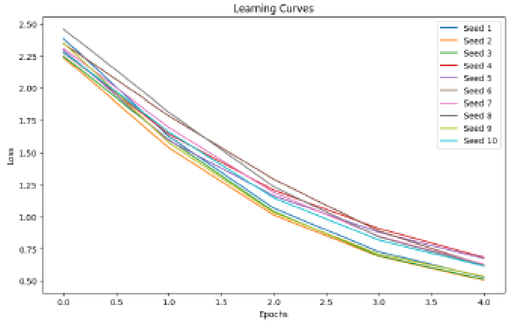
(a) Epoch 5 Context Size 1 RNN Layer 1



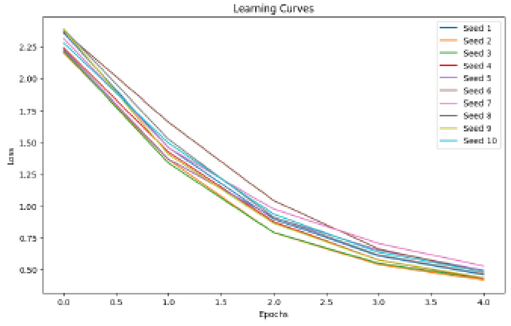
(b) Epoch 5 Context Size 2 RNN Layer 1



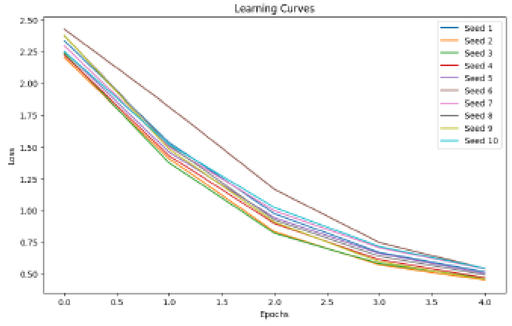
(c) Epoch 5 Context Size 1 RNN Layer 1



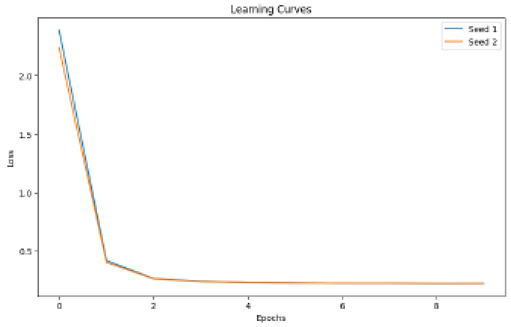
(d) Epoch 50 Context Size 1 RNN Layer 1



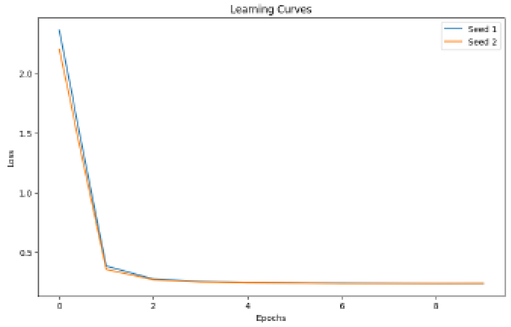
(e) Epoch 50 Context Size 2 RNN Layer 1



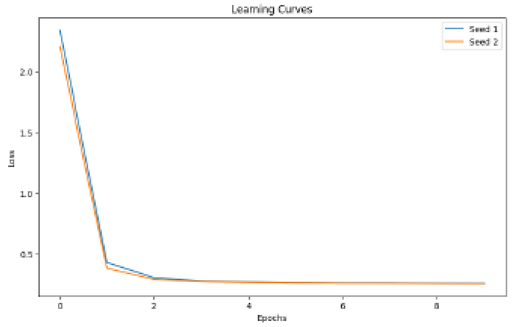
(f) Epoch 50 Context Size 3 RNN Layer 1



(g) Epoch 500 Context Size 1 RNN Layer 1



(h) Epoch 500 Context Size 2 RNN Layer 1



(i) Epoch 500 Context Size 3 RNN Layer 1

Efficient analysis

Efficient analysis

1. **Experiment**: Run a simple Artificial Neural Network (ANN) model 10 times using different random seeds. Use seeds ranging from 1 to 10 for these runs

   Question:

   1.1 Result Analysis: Record the output of each trial. Present this data in a tabular format, summarizing the convergence value for each seed. (1pt)

   Answer:

seeds/epoch	0	50	100	150	200	250	300
1	2.337935	0.354601	0.407567	0.393525	0.210256	0.379416	0.209338
2	2.305132	0.481161	0.220649	0.214474	0.214594	0.208037	0.209025
3	2.305132	0.481161	0.220649	0.214474	0.214594	0.208037	0.212019
4	2.293585	0.368123	0.421418	0.403463	0.401345	0.417959	0.210538
5	2.359769	0.418165	0.499559	0.217204	0.215842	0.210331	0.392910
6	2.310882	0.373581	0.227843	0.214972	0.210251	0.209413	0.208573
7	2.279085	0.340240	0.234737	0.415527	0.214810	0.214058	0.208971
8	2.310026	0.366771	0.222349	0.220332	0.382671	0.429902	0.212254
9	2.314622	0.302922	0.218075	0.213723	0.210246	0.449849	0.214507
10	2.307677	0.615157	0.227124	0.216396	0.210159	0.210810	0.214379

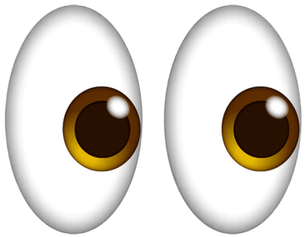
   1.2 Convergence Assessment: Analyze the convergence of the model across the different trials. Do the trials converge to similar values? Describe any patterns or variations you observe. (1pt)

seeds/delta	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6
1	-1.98333	0.05297	-0.01404	-0.18327	0.16916	-1.7008
2	-1.82397e+00	-2.60512e-01	-6.17474e-03	1.20007e-04	-6.55782e-03	9.88472e-04
3	-1.92546	0.05329	-0.01795	-0.00212	0.01661	-0.20594
4	-1.86122e+00	-2.07326e-01	-8.09717e-03	2.36416e-01	-2.39698e-01	1.15305e-03
5	-1.94160e+00	8.13943e-02	-2.82355e-01	-1.36128e-03	-5.51175e-03	1.82579e-01
6	-1.93730e+00	-1.45739e-01	-1.28702e-02	-4.72132e-03	-8.38023e-04	-8.39703e-04
7	-1.93884e+00	-1.05503e-01	1.80790e-01	-2.00717e-01	-7.52437e-04	-5.08620e-03
8	-1.94326	-0.14442	-0.00202	0.16234	0.04723	-0.21765
9	-2.0117	-0.08485	-0.00435	-0.00348	0.2396	-0.23534
10	-1.69252e+00	-3.88033e-01	-1.07279e-02	-6.23736e-03	6.51025e-04	3.56926e-03

   a) **Stricly decreasing loss** (seed 6) this is rare in the pattern where each 50 epoch the loss are stricly decreasing

Efficient analysis,

BUT



context size/layer	1-layer	2-layer
1	2.14328 0.96867 0.21449	2.19803 0.84945 0.21321
2	2.10417 0.73292 0.21363	2.18303 0.64828 0.21242
3	2.15295 0.94749 0.21578	2.20717 0.71965 0.21740

Efficient analysis

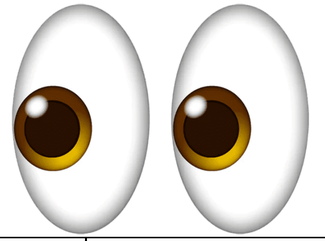
Seed	Convergence value
1	0.20890901506118098
2	0.4231551499815396
3	0.21043686416851415
4	0.2071146579537526
5	0.20788594314035436
6	0.20740645300702454
7	0.2088691889484835
8	0.2075280760291207
9	0.20700230367583572
10	0.38156100973355933

Epoch	Seed 9	Seed 10
0	2.390365242958069	2.2655791640281677
100	0.22030762874055654	0.22499913768842816
200	0.21000405805534683	0.2095427479071077
300	0.21580489291227423	0.2154759867116809
400	0.2141635422303807	0.4257956206238305
500	0.20866940270207124	0.2104116845766839
600	0.20719025186372164	0.20801487769404048
700	0.20702997776243137	0.20827203688349982
800	0.3866948010545457	0.20874753170755866
900	0.4437095504545141	0.2072629575195606

The model was trained for 1000 epochs for each seed!

Context Size	Epoch	Single-layer	Two-layer
1	5	2.074233293533325	2.13158056139946
	50	0.7587196081876755	0.9129638597369194
	500	0.43545057042501867	0.21326245518866926
2	5	2.2265852093696594	2.1011780500411987
	50	0.9571227729320526	0.7824751734733582
	500	0.21536519180517644	0.21267425164114684
3	5	2.231376826763153	2.088382303714752
	50	1.0131530165672302	0.6930344104766846
	500	0.4295507249189541	0.21611500484868884

Efficient analysis, BUT



Insufficient explanation:

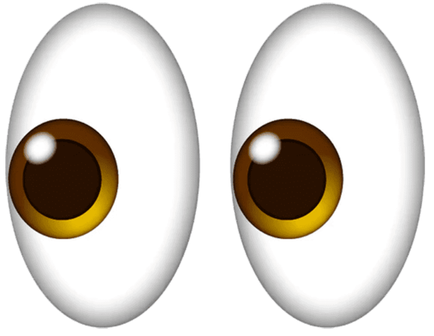
Below the different final loss values of the two models under the different configurations are tabulated. Again, there appear to be (seemingly random) spikes in the loss, skewing some of the final numbers (e.g. the single-layer RNN with context size 1 at epoch 500). In general, one can observe that the two-layer RNN converges faster than the single-layer RNN, which can be seen by the generally lower loss numbers at the different epoch checkpoints. Again, the performance of the two models appears to be very similar, as the final loss values given enough training epochs for the models to stabilize are in the same ballpark (ignoring the loss-spike outliers).

Context Size	Epoch	Single-layer	Two-layer
1	5	2.074233293533325	2.13158056139946
	50	0.7587196081876755	0.9129638597369194
	500	0.43545057042501867	0.21326245518866926
2	5	2.2265852093696594	2.1011780500411987
	50	0.9571227729320526	0.7824751734733582
	500	0.21536519180517644	0.21267425164114684
3	5	2.231376826763153	2.088382303714752
	50	1.0131530165672302	0.6930344104766846
	500	0.4295507249189541	0.21611500484868884

Efficient analysis

Epoch	seed	1	2	3	4	5	6	7	8	9	10
	Loss										
0		2.250562	2.309596	2.310017	2.301261	2.302918	2.360809	2.324537	2.357859	2.362674	2.306191
50		0.392268	0.813895	0.375627	0.344981	0.287587	0.332256	0.365583	0.333137	0.327558	0.738476
100		0.421231	0.22274	0.425717	0.224809	0.476357	0.227481	0.229492	0.223533	0.219012	0.252909
150		0.389039	0.220908	0.400565	0.213235	0.21254	0.216056	0.417303	0.218907	0.213099	0.219485
200		0.211231	0.214855	0.403019	0.445699	0.215294	0.210525	0.213968	0.38604	0.210406	0.211592
250		0.377163	0.209203	0.416664	0.209577	0.208619	0.208973	0.211894	0.433757	0.450292	0.210555
300		0.211078	0.208561	0.212548	0.20985	0.212243	0.207992	0.207702	0.407336	0.209774	0.210276
Mean		0.60751	0.599966	0.649165	0.564202	0.559365	0.537728	0.567211	0.622939	0.570402	0.592783
Standard Deviation		0.729798	0.786214	0.73601	0.771272	0.774911	0.805133	0.779493	0.769751	0.795449	0.779912

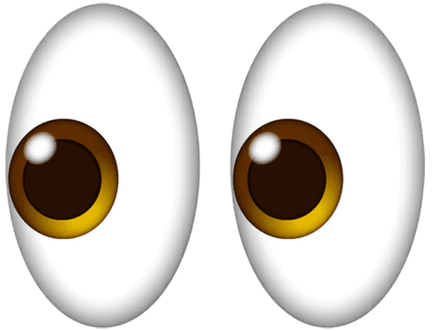
Efficient analysis, BUT



RNN Single Layer									
	Seeds								
	1	2	3	1	2	3	1	2	3
Epoch	Epoch=5			Epoch= 50			Epoch = 500		
0	2.271261	2.225614	2.206656	2.271261	2.225614	2.206656	2.271261	2.225614	0.280636
50							0.928347	0.79596	0.93969
100							0.450512	0.375656	0.433293
150							0.325132	0.283126	0.298391
200							0.257137	0.321303	0.262342
250							0.262109	0.237981	0.241299
300							0.22901	0.299344	0.239319
350							0.247722	0.222976	0.231752
400							0.218988	0.285835	0.223457
450							0.239514	0.280636	0.218753

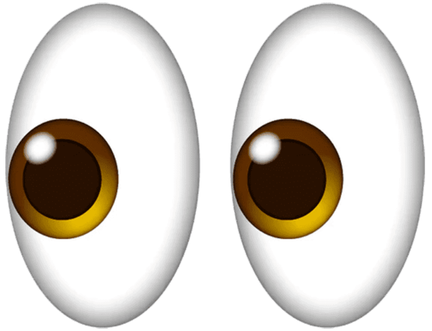
RNN Two Layers									
	Seeds								
	1	2	3	1	2	3	1	2	3
Epoch	Epoch=5			Epoch= 50			Epoch = 500		
0	2.345808	2.429621	2.358755	2.345808	2.429621	2.358755	2.345808	2.429621	2.358755
50							0.858577	0.745457	0.912622
100							0.42024	0.424309	0.60189
150							0.309305	0.338209	0.29649
200							0.269708	0.246435	0.256408
250							0.255865	0.301965	0.249871
300							0.224698	0.298473	0.23149
350							0.243127	0.222089	0.226583
400							0.217167	0.289807	0.225722
450							0.260636	0.285652	0.221843

Inefficient analysis



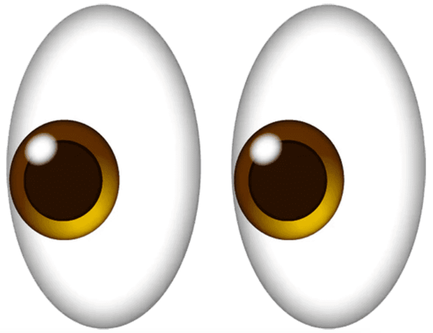
RNN Two Layers									
	Context Size								
	1	2	3	1	2	3	1	2	3
Epoch	Epoch=5			Epoch= 50			Epoch = 500		
0	2.26911	2.3082	2.30663	2.26911	2.3082	2.30663	2.229665	2.224899	2.217418
50							0.903053	0.728626	0.929122
100							0.460201	0.343818	0.369993
150							0.324266	0.269361	0.274288
200							0.254939	0.315108	0.251565
250							0.261164	0.232641	0.235451
300							0.227885	0.296214	0.229151
350							0.24707	0.220701	0.224905
400							0.21833	0.284184	0.220698
450							0.239031	0.278833	0.217639

Inefficient analysis



	1	2	3	1	2	3	1	2	3
Epoch	5			50			500		
0	2.2823	2.2774	2.2697	2.2823	2.2774	2.2697	2.2823	2.2774	2.2697
50							0.9244	0.7458	0.9510
100							0.4711	0.3519	0.3787
150							0.3319	0.2757	0.2808
200							0.2610	0.3225	0.2575
250							0.2673	0.2381	0.2410
300							0.2333	0.3032	0.2346
350							0.2529	0.2259	0.2302
400							0.2235	0.2909	0.2259
450							0.2447	0.2854	0.2228

Inefficient analysis



Seed 1

Epoch	Loss
0	2.366192341
50	0.3493744247
100	0.4063762077
150	0.3954754182
200	0.2097156881
250	0.3869852128

Seed 2

Epoch	Loss
0	2.331405044
50	0.3372015581
100	0.4120236321
150	0.3939476184
200	0.2096172672
250	0.3772637484

Seed 3

Epoch	Loss
0	2.295583367
50	0.7595753036
100	0.4115001909
150	0.3977958594
200	0.2096661173
250	0.3791766115

Seed 4

Epoch	Loss
0	2.310844123
50	0.8047514986
100	0.4049292845
150	0.3895440777
200	0.2113629534
250	0.380491803

Seed 5

Epoch	Loss
0	2.237056494
50	0.5515115689
100	0.4162336302
150	0.3884115909
200	0.2117844466
250	0.3781605641

Seed 6

Epoch	Loss
0	2.35028863
50	0.480442917
100	0.4213508624
150	0.3935596241
200	0.210971545
250	0.3776247652

Seed 7

Epoch	Loss
0	2.333400071
50	0.3863840178
100	0.4094354946
150	0.3867507987
200	0.2100818974
250	0.3771966391

Seed 8

Epoch	Loss
0	2.360630572
50	0.3618844561
100	0.4091057985
150	0.401258089
200	0.2100083014
250	0.3824582232

Seed 9

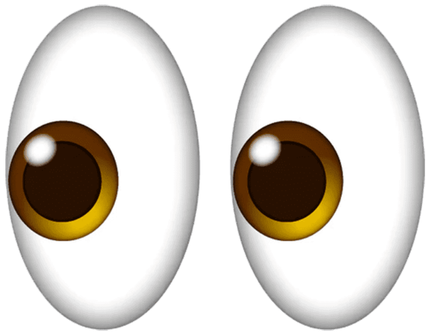
Epoch	Loss
0	2.327547908
50	0.3666462377
100	0.4065434379
150	0.3911204764
200	0.2102771586
250	0.3774728327

Seed 10

Epoch	Loss
0	2.289605737
50	0.660589842
100	0.4235699205
150	0.3897098645
200	0.2102928861
250	0.3771556436

Seed	Epoch and loss
1	Seed: 1, Epoch 0, Loss: 2.277082324028015 Seed: 1, Epoch 50, Loss: 0.42574357986450195 Seed: 1, Epoch 100, Loss: 0.421097198035568 Seed: 1, Epoch 150, Loss: 0.3901884563965723 Seed: 1, Epoch 200, Loss: 0.21025813436426688 Seed: 1, Epoch 250, Loss: 0.378179976134561
2	Seed: 2, Epoch 0, Loss: 2.3063393235206604 Seed: 2, Epoch 50, Loss: 0.6517390757799149 Seed: 2, Epoch 100, Loss: 0.230108039570041 Seed: 2, Epoch 150, Loss: 0.21931840060278773 Seed: 2, Epoch 200, Loss: 0.21613736409926787

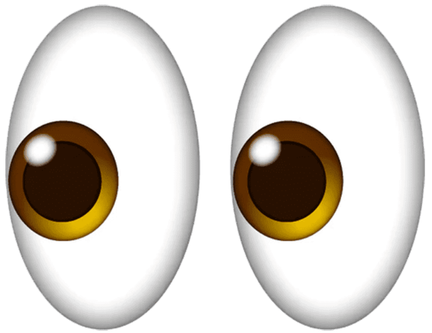
Inefficient analysis



```
Two-Layer RNN - Context_size 1
Epoch 0, Loss: 2.3135658502578735
Epoch 100, Loss: 0.35075761936604977
Epoch 200, Loss: 0.26616784324869514
Epoch 300, Loss: 0.22455711662769318
Epoch 400, Loss: 0.21807817555963993
Epoch 500, Loss: 0.2139274178771302

Two-Layer RNN - Context_size 2
Epoch 0, Loss: 2.316637694835663
Epoch 100, Loss: 0.3988543972373009
Epoch 200, Loss: 0.24275918491184711
Epoch 300, Loss: 0.296204190235585
Epoch 400, Loss: 0.288935349788516 76
Epoch 500, Loss: 0.212825430382508 78

Two-Layer RNN - Context_size 3
Epoch 0, Loss: 2.31792414188385
Epoch 100, Loss: 0.5249380543828011
Epoch 200, Loss: 0.2436244748532772
Epoch 300, Loss: 0.22763536032289267
Epoch 400, Loss: 0.22109109163284302
Epoch 500, Loss: 0.21633976546581835
```

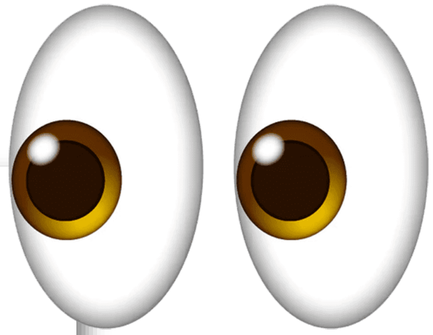
```
One-Layer RNN - Context_size 1
Epoch 0, Loss: 2.338731288909912
Epoch 100, Loss: 0.46079135686159134
Epoch 200, Loss: 0.25657771062105894
Epoch 300, Loss: 0.22828044975176454
Epoch 400, Loss: 0.21852817526087165
Epoch 500, Loss: 0.21451999840792269

One-Layer RNN - Context_size 2
Epoch 0, Loss: 2.373661458492279
Epoch 100, Loss: 0.36121646128594875
Epoch 200, Loss: 0.31832378543913364
Epoch 300, Loss: 0.2974740033969283
Epoch 400, Loss: 0.2854223665781319
Epoch 500, Loss: 0.2145187045680359

One-Layer RNN - Context_size 3
Epoch 0, Loss: 2.2913882732391357
Epoch 100, Loss: 0.39688240736722946
Epoch 200, Loss: 0.2580934129655361
Epoch 300, Loss: 0.2358868634328246
Epoch 400, Loss: 0.2225606068968773
Epoch 500, Loss: 0.2166112286504358
```

Inefficient analysis



```
[6]: {(1, 1, 5): [(5, 2.1433279514312744)],
      (2, 1, 5): [(5, 2.106285572052002)],
      (1, 2, 5): [(5, 2.143039643764496)],
      (2, 2, 5): [(5, 2.1761690378189087)],
      (1, 3, 5): [(5, 2.1027100682258606)],
      (2, 3, 5): [(5, 2.2250617146492004)],
      (1, 1, 50): [(5, 2.232255697250366), (50, 1.033349484205246)],
      (2, 1, 50): [(5, 2.208823502063751), (50, 0.9524664729833603)],
      (1, 2, 50): [(5, 2.232119083404541), (50, 0.8465640842914581)],
      (2, 2, 50): [(5, 2.177534282207489), (50, 0.6457824185490608)],
      (1, 3, 50): [(5, 2.1164112985134125), (50, 0.8020300567150116)],
      (2, 3, 50): [(5, 2.2590193152427673), (50, 0.6749854385852814)],
      (1, 1, 500): [(5, 2.1699026823043823),
       (50, 1.036008283495903),
       (500, 0.2372704908484593)],
      (2, 1, 500): [(5, 2.141894668340683),
       (50, 0.8046733140945435),
       (500, 0.23638491320889443)],
      (1, 2, 500): [(5, 2.1420746445655823),
       (50, 0.8469626009464264),
       (500, 0.21567761828191578)],
      (2, 2, 500): [(5, 2.1866953372955322),
       (50, 0.7606131732463837),
       (500, 0.21514265867881477)],
      (1, 3, 500): [(5, 2.189723312854767),
       (50, 0.8189168646931648),
       (500, 0.43254566565155983)],
      (2, 3, 500): [(5, 2.1953893303871155),
       (50, 0.8300317898392677),
       (500, 0.21583306440152228)]}
```

Inefficient analysis

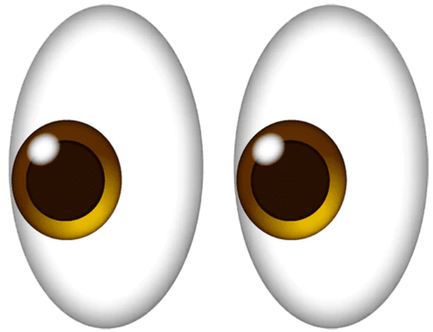


2.1. Convergence Analysis: At which epoch numbers do the single-layer and two-layer RNNs begin to converge for each context size? Record the earliest epoch at which each configuration appears to stabilize.

Singe Layer
epoch=5
Epoch 0, Loss: 2.4071923891703286
Epoch 5, Loss: 2.238239006221517
Epoch 0, Loss: 2.3217293421427407
Epoch 5, Loss: 2.110893408457438
Epoch 0, Loss: 2.368868360619409
Epoch 5, Loss: 2.2100552320480347
Epoch 0, Loss: 2.3381643295288086
Epoch 50, Loss: 1.4838362137476604
Epoch 0, Loss: 2.1642823219299316
Epoch 50, Loss: 1.0352608561515608
Epoch 0, Loss: 2.3986504077911377
Epoch 50, Loss: 1.1980800032615662
Epoch 0, Loss: 2.359814167022705
Epoch 50, Loss: 1.3953742186228435
Epoch 100, Loss: 0.7292661468187968
Epoch 150, Loss: 0.3805410961310089
Epoch 200, Loss: 0.259467676261929
Epoch 250, Loss: 0.2020411867764931
Epoch 300, Loss: 0.1743198757370313
Epoch 350, Loss: 0.16046677281459173
Epoch 400, Loss: 0.14842802720765272
Epoch 450, Loss: 0.14120366803440413
Epoch 500, Loss: 0.25261090695858
Epoch 0, Loss: 2.344271977742513
Epoch 50, Loss: 1.0354307691256206
Epoch 100, Loss: 0.3816165127334595

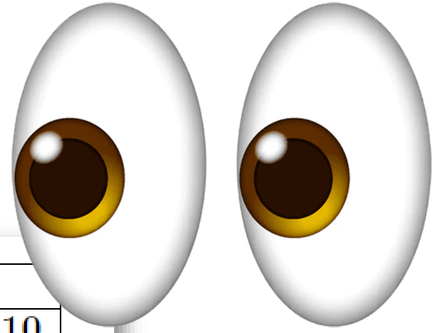
Multiple Layer
epoch=5
Epoch 0, Loss: 2.4539171854654946
Epoch 5, Loss: 2.320923328399658
Epoch 0, Loss: 2.3665260473887124
Epoch 5, Loss: 2.216878811518351
Epoch 0, Loss: 2.437862515449524
Epoch 5, Loss: 2.2709032353591192
Epoch 0, Loss: 2.3589882986450195
epochs=51
Epoch 0, Loss: 2.2303890387217202
Epoch 50, Loss: 1.2447996536890686
Epoch 0, Loss: 2.4202037652333 58
Epoch 50, Loss: 0.9740008910497 03
Epoch 0, Loss: 2.3068239688873 29
Epoch 50, Loss: 0.8759709596633911
epochs=501
Epoch 0, Loss: 2.343861738840739
Epoch 50, Loss: 1.263209382692973
Epoch 100, Loss: 0.4458787242571513
Epoch 150, Loss: 0.2659068455298741 6
Epoch 200, Loss: 0.1895128265023231 5
Epoch 250, Loss: 0.2340445491525778
Epoch 300, Loss: 0.1486784958591064
Epoch 350, Loss: 0.1945813434819380 3
Epoch 400, Loss: 0.1354030330354968 6
Epoch 450, Loss: 0.1904720167318979 8
Epoch 500, Loss: 0.1279796501621604
Epoch 0, Loss: 2.343861738840739
Epoch 50, Loss: 1.263209382692973
Epoch 100, Loss: 0.4458787242571513
Epoch 150, Loss: 0.2659068455298741 5
Epoch 200, Loss: 0.1895128265023231 5

Inefficient analysis



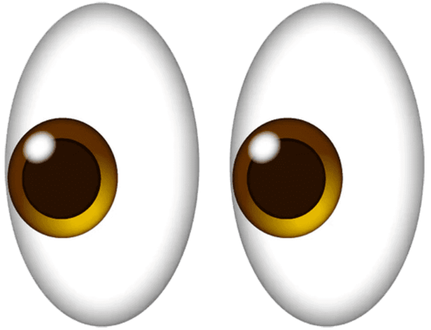
Single Layer RNN				Two Layer RNN			
Epoch	5	50	500	Epoch	5	50	500
	2.1942	1.072	0.2382		2.2572	0.9382	0.2372
	2.1381	0.7921	0.2151		2.2243	0.6461	0.2131
	2.083	0.772	0.2163		2.2231	0.6332	0.2163

Inefficient analysis



Epoch	Loss									
	Seed 1	Seed 2	Seed 3	Seed 4	Seed 5	Seed 6	Seed 7	Seed 8	Seed 9	Seed 10
0	2.292	2.307	2.273	2.294	2.349	2.369	2.310	2.358	2.241	2.333
50	0.267	0.276	0.272	0.279	0.285	0.268	0.271	0.303	0.272	0.279
100	0.259	0.264	0.264	0.263	0.272	0.261	0.262	0.261	0.259	0.259
150	0.256	0.263	0.260	0.261	0.260	0.267	0.262	0.261	0.259	0.262
200	0.256	0.261	0.261	0.258	0.260	0.257	0.257	0.259	0.261	0.258
250	0.258	0.260	0.261	0.259	0.259	0.257	0.260	0.259	0.259	0.263
300	0.258	0.262	0.256	0.259	0.262	0.258	0.260	0.261	0.259	0.260
350	0.258	0.258	0.260	0.259	0.259	0.256	0.258	0.259	0.259	0.255
400	0.258	0.257	0.258	0.257	0.256	0.258	0.256	0.256	0.256	0.260
450	0.256	0.256	0.259	0.256	0.256	0.258	0.258	0.257	0.255	0.260
500	0.256	0.257	0.256	0.259	0.257	0.256	0.257	0.257	0.255	0.257
550	0.256	0.256	0.256	0.255	0.257	0.257	0.256	0.257	0.257	0.256
600	0.257	0.256	0.259	0.256	0.256	0.256	0.257	0.255	0.257	0.259
650	0.255	0.257	0.256	0.258	0.255	0.258	0.256	0.256	0.255	0.256
700	0.254	0.255	0.256	0.257	0.258	0.256	0.256	0.254	0.255	0.257
750	0.256	0.254	0.256	0.256	0.256	0.256	0.256	0.256	0.255	0.256
800	0.256	0.255	0.255	0.256	0.257	0.256	0.256	0.256	0.255	0.255
850	0.256	0.254	0.255	0.255	0.256	0.256	0.255	0.255	0.256	0.256
900	0.255	0.256	0.255	0.256	0.255	0.255	0.256	0.256	0.255	0.256
950	0.255	0.254	0.254	0.256	0.255	0.255	0.255	0.255	0.255	0.255

Inefficient analysis



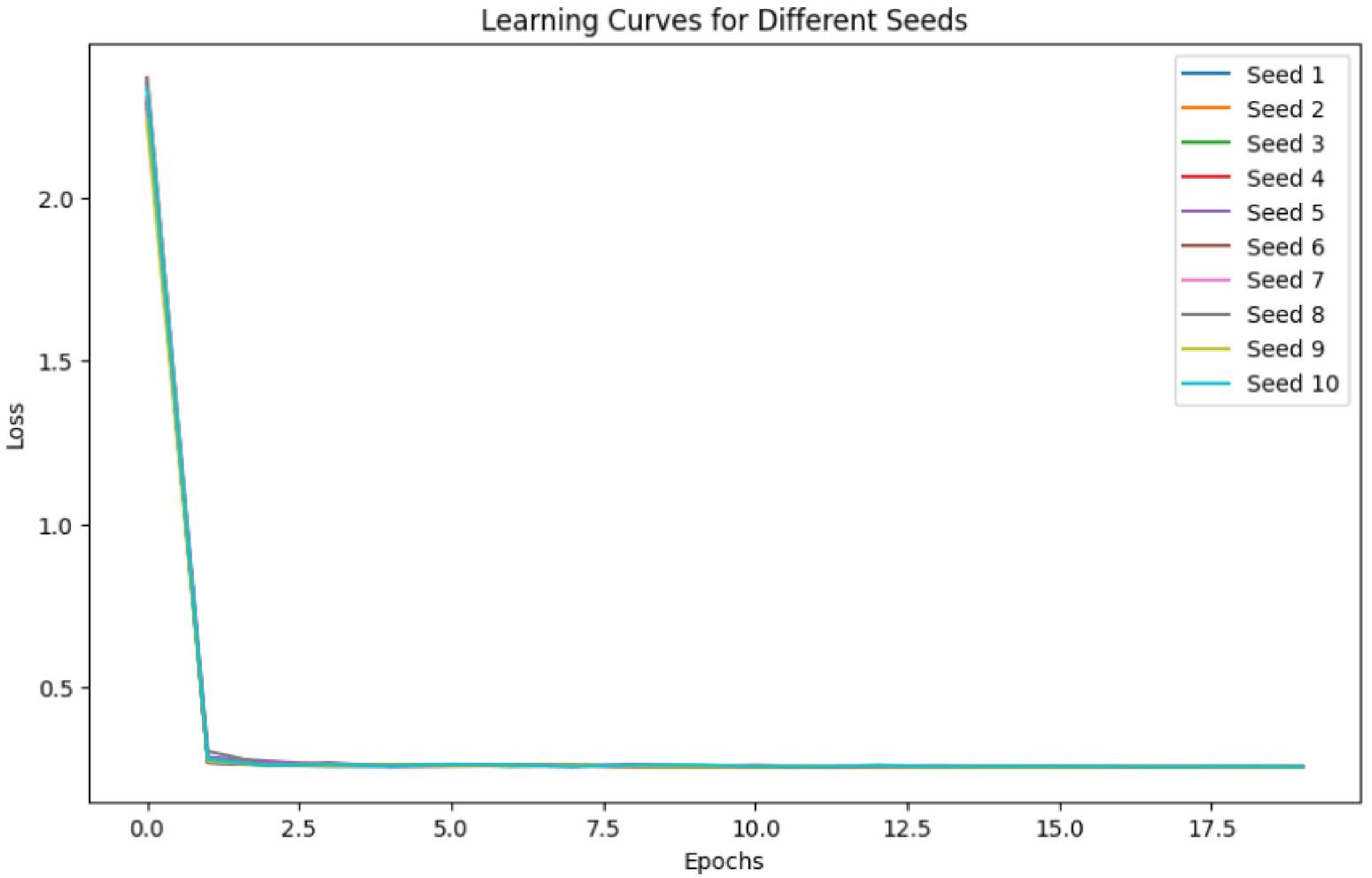
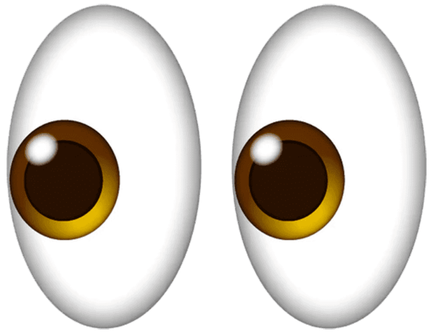


Figure 1

Inefficient analysis



We can calculate the mean as follows:

$$\begin{aligned}\text{Mean} &= \frac{0.25451979270327085 + 0.25446116924251344 + 0.25422236552564326}{10} \\ &+ \frac{0.25468811621947435 + 0.2552229991320057 + 0.25524405332668737}{10} \\ &+ \frac{0.255103 + 0.254989 + 0.2544695 + 0.255064}{10} \\ &\approx 0.25498334166664824\end{aligned}$$

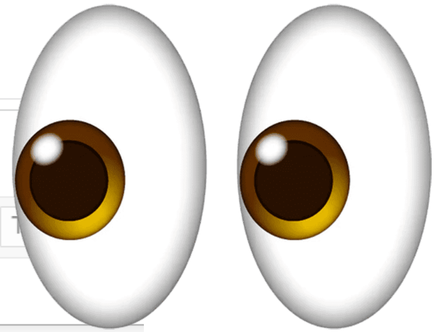
Now, let's calculate the standard deviation:

$$\begin{aligned}\text{Standard Deviation} &= \sqrt{\frac{(0.25452 - \text{Mean})^2 + \cdots + (0.25506 - \text{Mean})^2}{10}} \\ &\approx \sqrt{\frac{0.000087^2 + \cdots + 0.00008086^2}{10}} \\ &\approx \sqrt{\frac{3.183506768142275 \times 10^{-9}}{10}} \\ &\approx \sqrt{3.183506768142275 \times 10^{-10}} \\ &\approx 0.0005642092244407737\end{aligned}$$

So, the manual calculations yield:

$$\begin{aligned}\text{Mean} &\approx 0.25498334166664824 \\ \text{Standard Deviation} &\approx 0.0005642092244407737\end{aligned}$$

???



jupyter HW_Tut_8 Last Checkpoint: 7 hours ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Run Markdown

In [32]:

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
```

Code Answer Number 1

In [33]:

```
#Preprocessing
text = "If you win, you live! If you lose, you die! If you don't fight, you can't win!"
sentences = text.lower().split(' ')

#Embedding
word_to_ix = {word: i for i, word in enumerate(set(sentences))}
vocab_size = len(word_to_ix)
embedding_dim = 100
embedding_matrix = nn.Parameter(torch.randn(vocab_size, embedding_dim), requires_grad=True)

#The model
class ANN(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(ANN, self).__init__()
        self.embedding = nn.Embedding.from_pretrained(embedding_matrix)
```

National Taiwan University of Science and Technology
Computer Science and Information Engineering

TUTORIAL 11: TRANSFORMER (PART 2)

Instructor: Prof. Hsing-Kuo Pao
TA: Zolnamar Dorjsembe (Zola)

Date: May 07, 2024

Solving Transformer by Hand: A Step-by-Step Math Example

Source: https://levelup.gitconnected.com/understanding-transformers-from-start-to-end-a-step-by-step-math-example-16d4e64e6eb1

Step 1. Defining Our Dataset

Dataset (corpus)

I drink and I know things.
When you play the game of thrones, you win or you die.
The true enemy won't wait out the storm, He brings the storm.

Our entire dataset containing only three sentences

Step 2. Finding Vocab Size

$$vocab\ size = count(set(N))$$

vocab_size formula where N is total number of words

Dataset (Corpus)

I drink and I know things.
When you play the game of thrones, you win or you die.
The true enemy won't wait out the storm, He brings the storm.

N = [
I, drink, and, I, Know, things,
When, you, play, the, game, of, thrones, you, win, or, you, die,
The, true, enemy, won't, wait, out, the, storm, He, brings, the, storm
]

calculating variable **N**

$$vocab\ size = count(set(N))$$

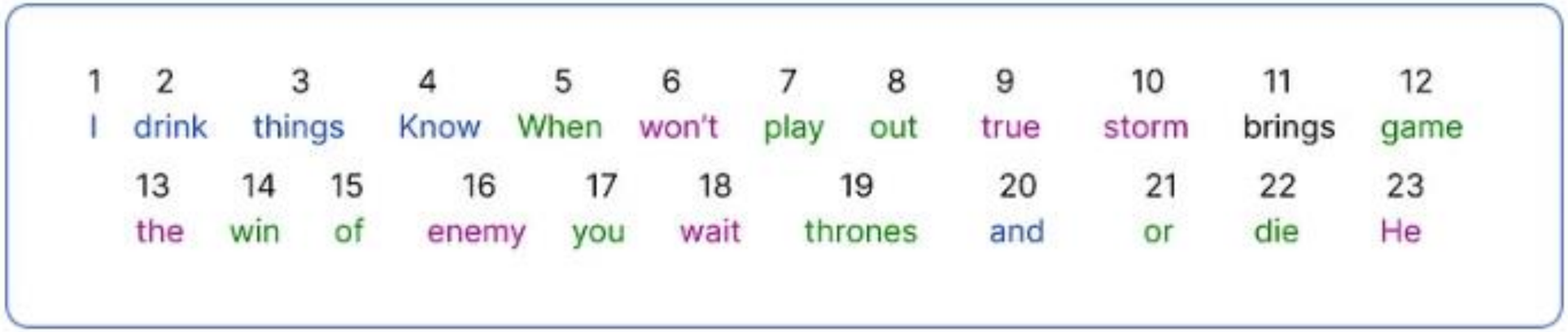
set (I, drink, and, I, Know, things, When, you, play, the, game, of, thrones, you, win, or, you, die, The, true, enemy, won't, wait, out, the, storm, He, brings, the, storm)

count (I, drink, and, Know, things, When, you, play, the, game, of, thrones, win, or, die, true, enemy, won't, wait, out, storm, He, brings)

= 23

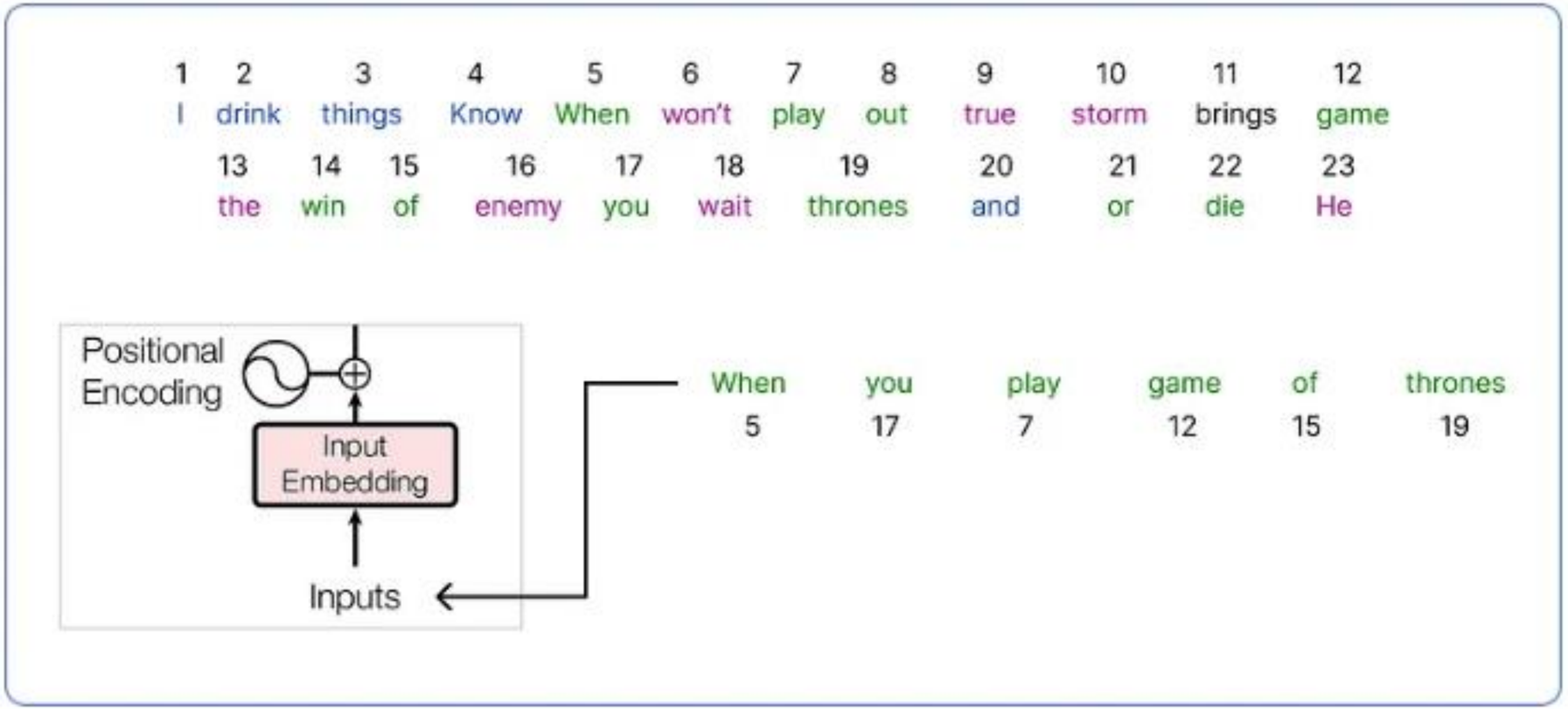
finding vocab size

Step 3. Encoding



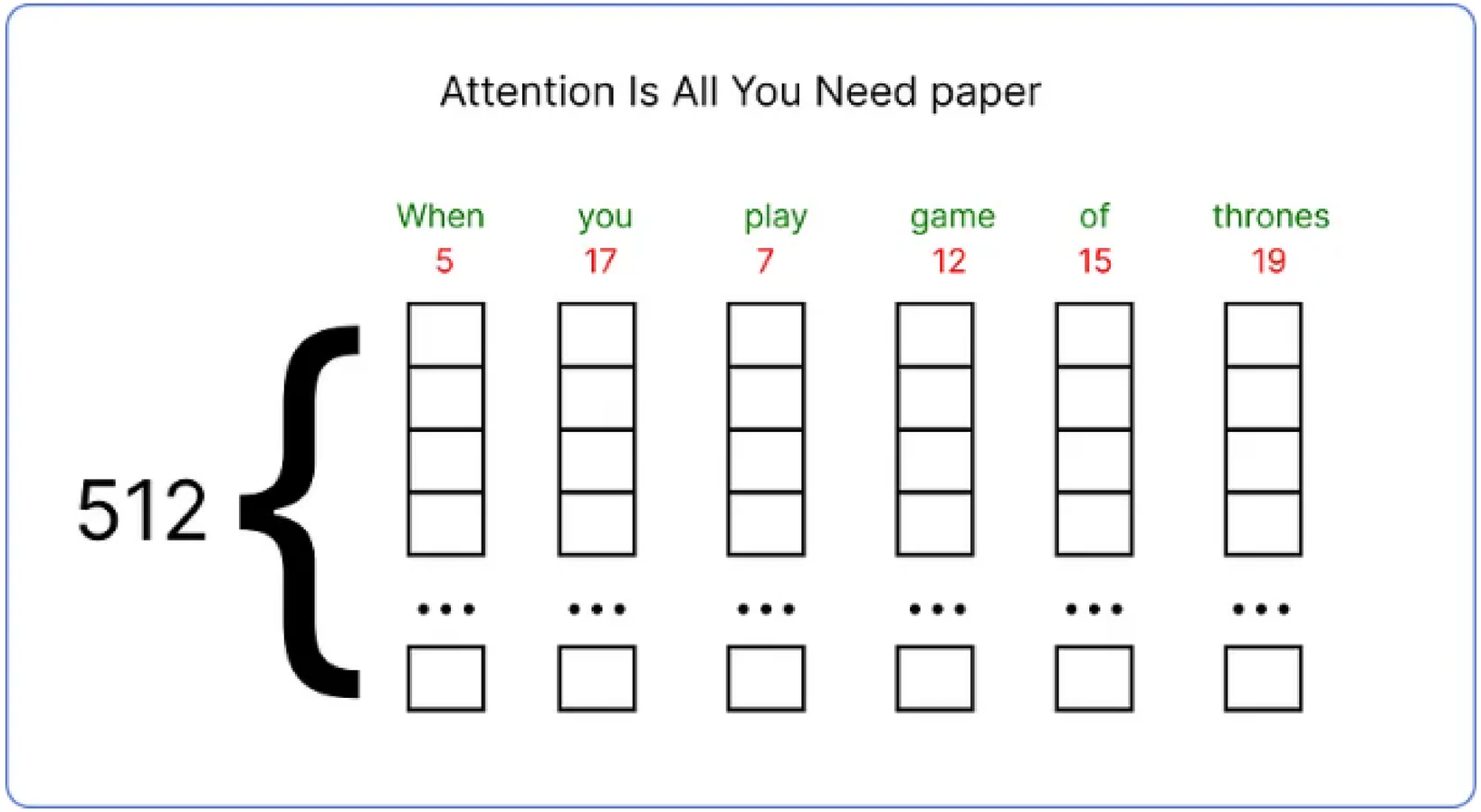
encoding our unique words

Step 4. Calculating Embedding



Input sentence for transformer

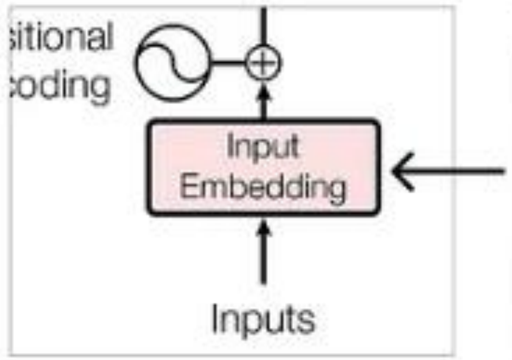
Step 4. Calculating Embedding



Original Paper uses 512 dimension vector

Step 4. Calculating Embedding

(For demonstration, use an embedding vector with a dimension of 6)



When	you	play	game	of	thrones
5	17	7	12	15	19

e1	e2	e3	e4	e5	e6
0.79	0.38	0.01	0.12	0.88	0.6
0.6	0.12	0.51	0.6	0.41	0.33
0.96	0.06	0.27	0.65	0.79	0.75
0.64	0.79	0.31	0.22	0.62	0.48
0.97	0.9	0.56	0.07	0.5	0.94
0.2	0.74	0.59	0.37	0.7	0.21

Embedding vectors of our input

Step 5. Calculating Positional Embedding

Embedding vector for any word

For even position

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

For odd position

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

Positional Embedding formula

Step 5. Calculating Positional Embedding

When
5

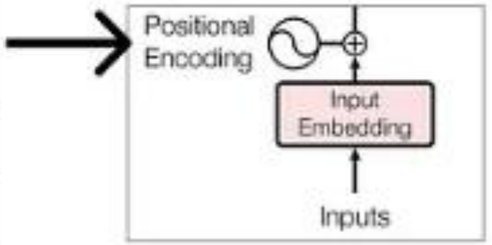
i	e1	Position	Formula	p1
0	0.79	Even	$sin(0/10000^{(2*0/6)})$	0
1	0.6	Odd	$cos(0/10000^{(2*1/6)})$	1
2	0.96	Even	$sin(0/10000^{(2*2/6)})$	0
3	0.64	Odd	$cos(0/10000^{(2*3/6)})$	1
4	0.97	Even	$sin(0/10000^{(2*4/6)})$	0
5	0.2	Odd	$cos(0/10000^{(2*5/6)})$	1
d (dim)	6			
POS	0			

Positional Embedding for word: **When**

Step 5. Calculating Positional Embedding

When	you	play	game	of	thrones
5	17	7	12	15	19

i	p1	p2	p3	p4	p5	p6
0	0	0.8415	0.9093	0.1411	-0.7568	-0.9589
1	1	0.0464	0.9957	0.1388	0.1846	0.9732
2	0	0.0022	0.0043	0.0065	0.0086	0.0108
3	1	0.0001	1	0.0003	0.0004	1
4	0	0	0	0	0	0
5	1	0	1	0	0	1
d (dim)	6	6	6	6	6	6
POS	0	1	2	3	4	5



Calculating Positional Embeddings of our input **(The calculated values are rounded)**

Step 6. Concatenating Positional and Word Embeddings

When	you	play	game	of	thrones
5	17	7	12	15	19

Position Embedding Matrix

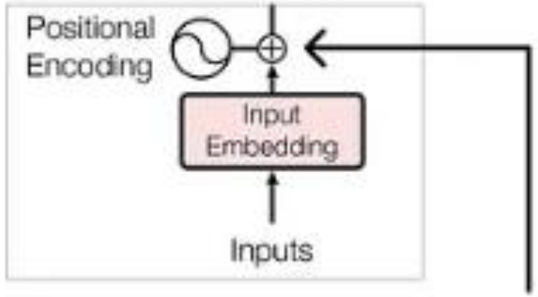
p1	p2	p3	p4	p5	p6
0	0.8415	0.9093	0.1411	-0.7568	-0.9589
1	0.0464	0.9957	0.1388	0.1846	0.9732
0	0.0022	0.0043	0.0065	0.0086	0.0108
1	0.0001	1	0.0003	0.0004	1
0	0	0	0	0	0
1	0	1	0	0	1

+

Word Embedding Matrix

e1	e2	e3	e4	e5	e6
0.79	0.38	0.01	0.12	0.88	0.6
0.6	0.12	0.51	0.6	0.41	0.33
0.96	0.06	0.27	0.65	0.79	0.75
0.64	0.79	0.31	0.22	0.62	0.48
0.97	0.9	0.56	0.07	0.5	0.94
0.2	0.74	0.59	0.37	0.7	0.21

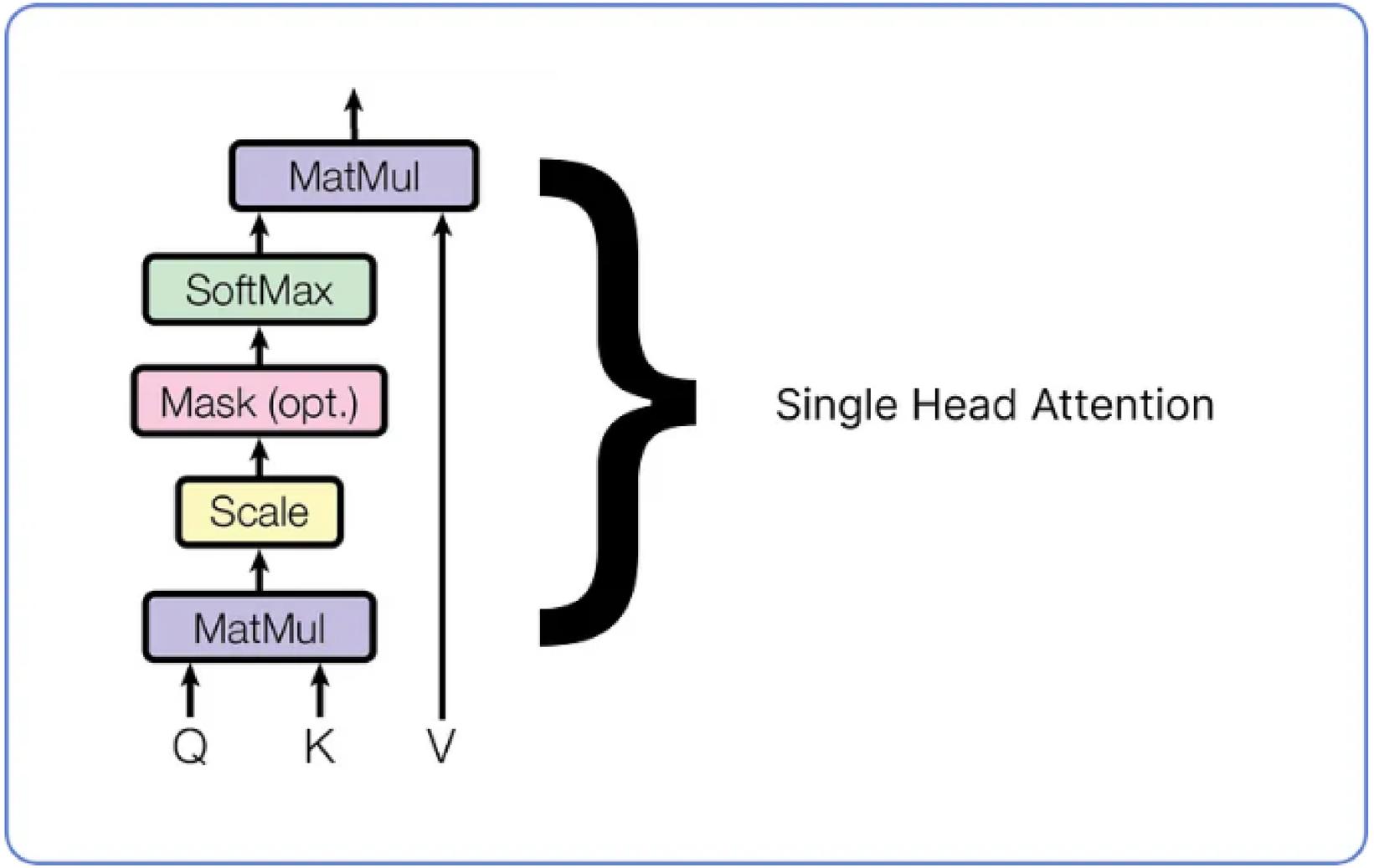
=



ep1	ep2	ep3	ep4	ep5	ep6
0.79	1.22	0.92	0.26	0.12	-0.36
1.6	0.17	1.51	0.74	0.59	1.3
0.96	0.06	0.27	0.66	0.8	0.76
1.64	0.79	1.31	0.22	0.62	1.48
0.97	0.9	0.56	0.07	0.5	0.94
1.2	0.74	1.59	0.37	0.7	1.21

concatenation step

Step 7. Multi Head Attention



Single Head attention in Transformer

Step 7. Multi Head Attention: Query matrix

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6 x 6

X

Linear weights for query

0.52	0.45	0.91	0.69
0.05	0.85	0.37	0.83
0.49	0.1	0.56	0.61
0.71	0.64	0.4	0.14
0.76	0.27	0.92	0.67
0.85	0.56	0.57	0.07

6 x 4

calculating Query matrix

Step 7. Multi Head Attention: Key and Value Matrices

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6 x 6

X

Linear weights for key

0.74	0.57	0.21	0.73
0.55	0.16	0.9	0.17
0.25	0.74	0.8	0.98
0.8	0.73	0.2	0.31
0.37	0.96	0.42	0.08
0.28	0.41	0.87	0.86

6 x 4

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6 x 6

X

Linear weights for value

0.62	0.07	0.7	0.95
0.2	0.97	0.61	0.35
0.57	0.8	0.61	0.5
0.67	0.35	0.98	0.54
0.47	0.83	0.34	0.94
0.6	0.69	0.13	0.98

6 x 4

Calculating Key and Value Matrices

Step 7. Multi Head Attention: Query, Key and Values

Query

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6 x 4

key

3.71	4.04	4.15	3.41
2.18	2.51	1.64	1.93
3.28	3.11	3.65	3.01
1.07	1.13	1.64	1.35
1.49	1.97	2.14	1.81
2.51	3.04	3.45	2.28

6 x 4

value

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6 x 4

Query, Key, Value matrices

Step 7. Multi Head Attention:
Calculating single-head attention

Query

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6 x 4

X

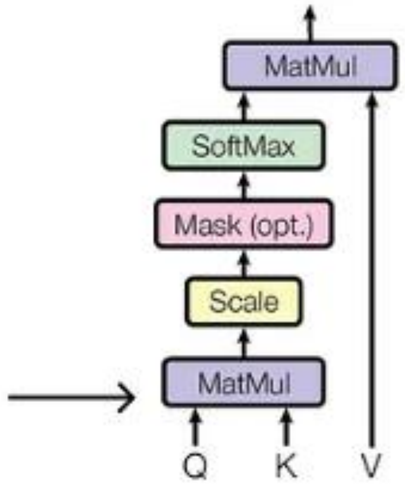
Transpose (Key)

3.71	2.18	3.28	1.07	1.49	2.51
4.04	2.51	3.11	1.13	1.97	3.04
4.15	1.64	3.65	1.64	2.14	3.45
3.41	1.93	3.01	1.35	1.81	2.28

4 x 6

=

58.341	31.2882	49.7306	19.7538	28.1886	43.1644
34.5402	18.2058	29.6169	11.7761	16.6133	25.6698
50.8796	27.3994	43.2409	17.0909	24.5349	37.695
18.1304	9.728	15.4544	6.2134	8.851	13.3894
26.3707	14.0937	22.5509	8.9445	12.6967	19.4858
41.8941	22.668	35.6369	14.004	20.103	30.9265



matrix multiplication between Query and Key

Step 7. Multi Head Attention:
Calculating single-head attention: WHY DOT PRODUCT?

Query

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6 x 4

X

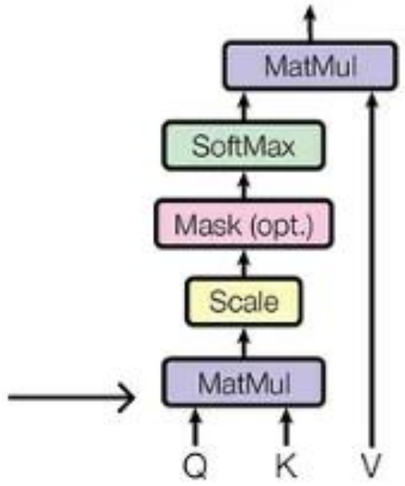
Transpose (Key)

3.71	2.18	3.28	1.07	1.49	2.51
4.04	2.51	3.11	1.13	1.97	3.04
4.15	1.64	3.65	1.64	2.14	3.45
3.41	1.93	3.01	1.35	1.81	2.28

4 x 6

=

58.341	31.2882	49.7306	19.7538	28.1886	43.1644
34.5402	18.2058	29.6169	11.7761	16.6133	25.6698
50.8796	27.3994	43.2409	17.0909	24.5349	37.695
18.1304	9.728	15.4544	6.2134	8.851	13.3894
26.3707	14.0937	22.5509	8.9445	12.6967	19.4858
41.8941	22.668	35.6369	14.004	20.103	30.9265



matrix multiplication between Query and Key

Step 7. Multi Head Attention:
Calculating single-head attention: WHY DOT PRODUCT?

Query

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6 x 4

X

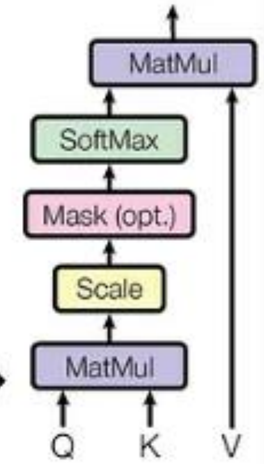
Transpose (Key)

3.71	2.18	3.28	1.07	1.49	2.51
4.04	2.51	3.11	1.13	1.97	3.04
4.15	1.64	3.65	1.64	2.14	3.45
3.41	1.93	3.01	1.35	1.81	2.28

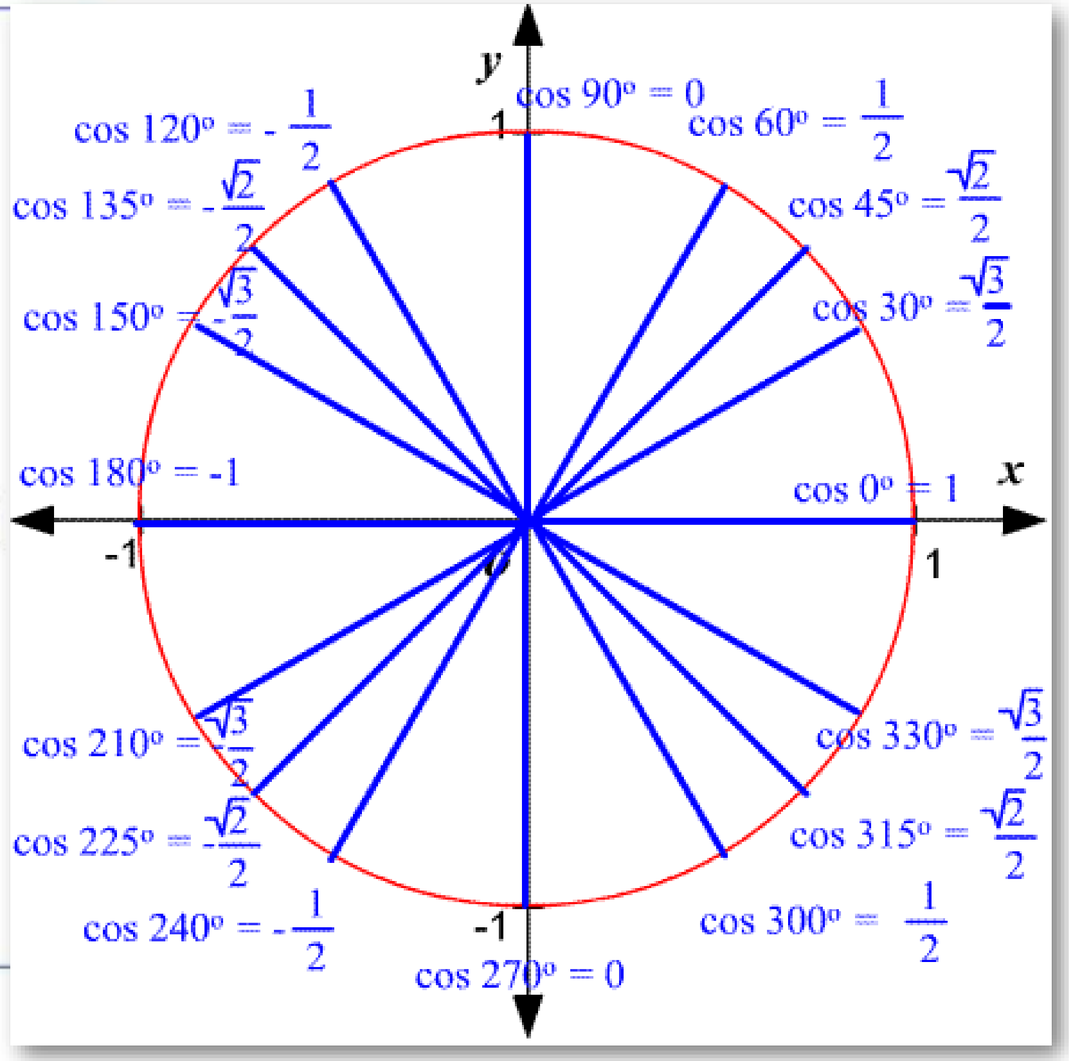
4 x 6

=

58.341	31.2882	49.7306	19.7538	28.1886	43.1644
34.5402	18.2058	29.6169	11.7761	16.6133	25.6698
50.8796	27.3994	43.2409	17.0909	24.5349	37.695
18.1304	9.728	15.4544	6.2134	8.851	13.3894
26.3707	14.0937	22.5509	8.9445	12.6967	19.4858
41.8941	22.668	35.6369	14.004	20.103	30.9265



matrix multiplication between Query and Key



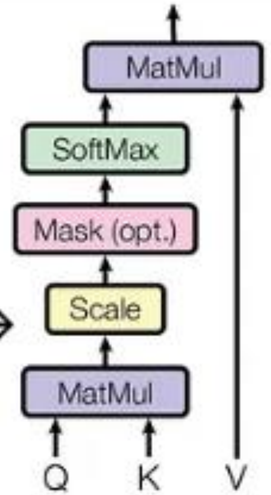
Step 7. Multi Head Attention:
Calculating single-head attention

58.341	31.2882	49.7306	19.7538	28.1886	43.1644
34.5402	18.2058	29.6169	11.7761	16.6133	25.6698
50.8796	27.3994	43.2409	17.0909	24.5349	37.695
18.1304	9.728	15.4544	6.2134	8.851	13.3894
26.3707	14.0937	22.5509	8.9445	12.6967	19.4858
41.8941	22.668	35.6369	14.004	20.103	30.9265

$\sqrt{d_k}$ where d (dimension) is 6

=

23.81721	12.77409	20.30219	8.062904	11.50852	17.62
14.1009	7.434201	12.09231	4.809165	6.781004	10.47973
20.77167	11.186	17.65266	6.976963	10.01433	15.39096
7.401542	3.972256	6.307436	2.535222	3.612997	5.466445
10.76551	5.752218	9.205999	3.64974	5.184753	7.956759
17.10152	9.254989	14.54997	5.715476	8.205791	12.62712



scaling the resultant matrix with dimension **5**

Step 7. Multi Head Attention:

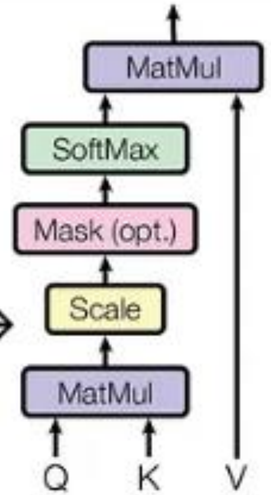
Calculating single-head attention: WHY $\sqrt{d_k}$?

58.341	31.2882	49.7306	19.7538	28.1886	43.1644
34.5402	18.2058	29.6169	11.7761	16.6133	25.6698
50.8796	27.3994	43.2409	17.0909	24.5349	37.695
18.1304	9.728	15.4544	6.2134	8.851	13.3894
26.3707	14.0937	22.5509	8.9445	12.6967	19.4858
41.8941	22.668	35.6369	14.004	20.103	30.9265

$\sqrt{d_k}$ where d (dimension) is 6

=

23.81721	12.77409	20.30219	8.062904	11.50852	17.62
14.1009	7.434201	12.09231	4.809165	6.781004	10.47973
20.77167	11.186	17.65266	6.976963	10.01433	15.39096
7.401542	3.972256	6.307436	2.535222	3.612997	5.466445
10.76551	5.752218	9.205999	3.64974	5.184753	7.956759
17.10152	9.254989	14.54997	5.715476	8.205791	12.62712



scaling the resultant matrix with dimension **5**

Step 7. Multi Head Attention:

Calculating single-head attention: WHY $\sqrt{d_k}$?

1. The numerical values should be kept at the same range
2. A vector product multiplies numerical values on average by the square root of vector dimension

Proof: https://github.com/BAI-Yeqi/Statistical-Properties-of-Dot-Product/blob/master/proof.pdf

17.10152	9.254989	14.54997	5.715476	8.205791	12.62712

scaling the resultant matrix with dimension **5**

Step 7. Multi Head Attention:
Calculating single-head attention

23.82	12.77	20.3	8.06	11.51	17.62
14.1	7.43	12.09	4.81	6.78	10.48
20.77	11.19	17.65	6.98	10.01	15.39
7.4	3.97	6.31	2.54	3.61	5.47
10.77	5.75	9.21	3.65	5.18	7.96
17.1	9.25	14.55	5.72	8.21	12.63

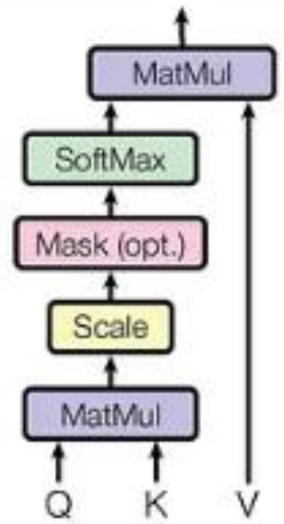
SoftMax

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

$$\text{softmax}(23.82) = \frac{e^{23.82}}{e^{23.82} + e^{12.77} + e^{20.3} + e^{8.06} + e^{11.51} + e^{17.62}}$$

=

0.9693	0	0.0287	0	0	0.002
0.86	0.0011	0.1152	0.0001	0.0006	0.023
0.9534	0.0001	0.0421	0	0	0.0044
0.6476	0.021	0.2177	0.005	0.0146	0.094
0.7803	0.0052	0.164	0.0006	0.0029	0.047
0.9174	0.0004	0.0716	0	0.0001	0.0105



Applying softmax on resultant matrix

Step 7. Multi Head Attention:
Calculating single-head attention

$$\text{softmax}\left(\frac{Q.K^T}{\sqrt{d_k}}\right)$$

0.9693	0	0.0287	0	0	0.002
0.86	0.0011	0.1152	0.0001	0.0006	0.023
0.9534	0.0001	0.0421	0	0	0.0044
0.6476	0.021	0.2177	0.005	0.0146	0.094
0.7803	0.0052	0.164	0.0006	0.0029	0.047
0.9174	0.0004	0.0716	0	0.0001	0.0105

6 x 6

X

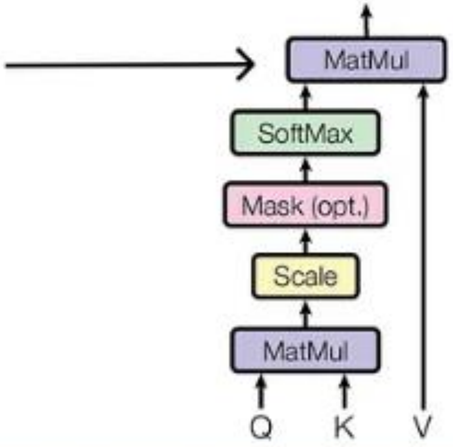
Value

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6 x 4

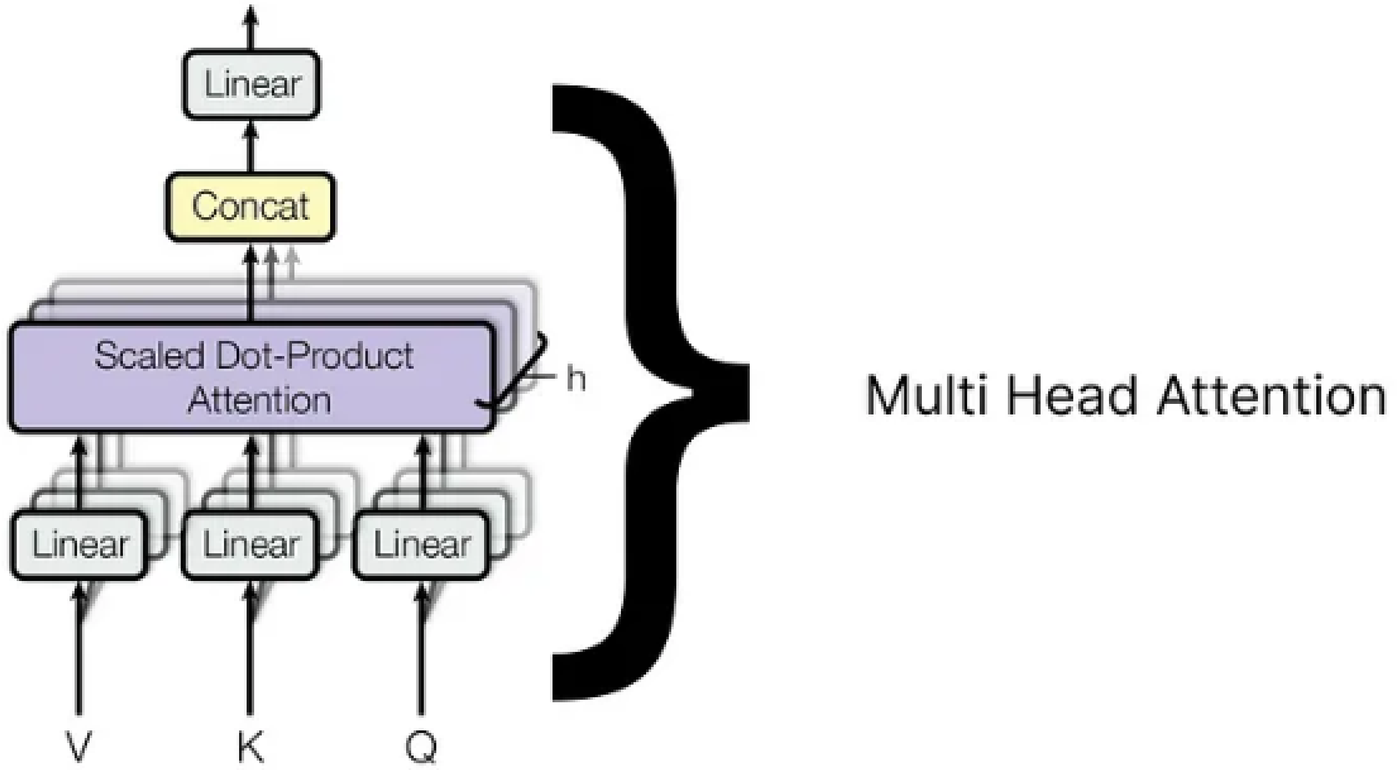
=

3.864257	3.79246	4.060367	3.39751
3.801295	3.75252	3.977937	3.30861
3.855542	3.787426	4.04909	3.385086
3.622841	3.584936	3.750419	3.081834
3.745786	3.706744	3.904894	3.233519
3.835366	3.77523	4.022837	3.356435



calculating the final matrix of single head attention

Step 7. Multi Head Attention:

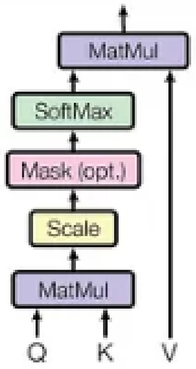


Multi Head attention in Transformer

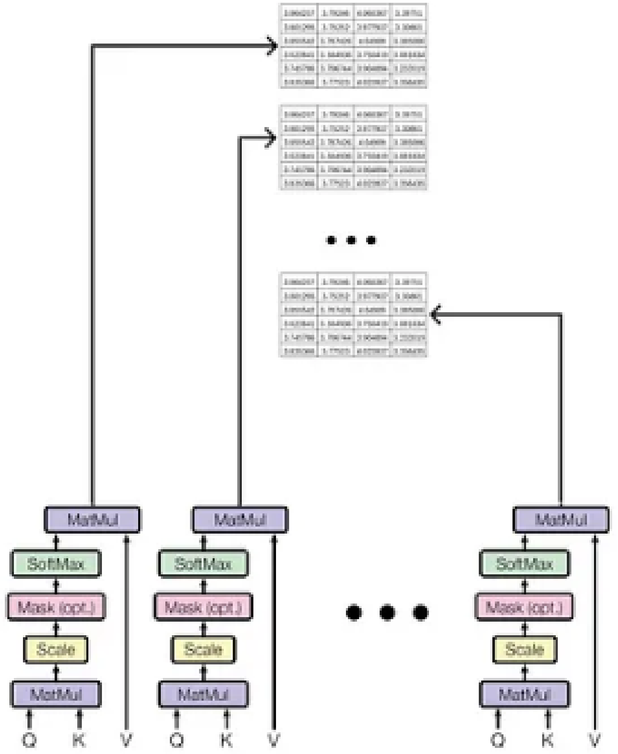
Step 7. Multi Head Attention:
Single-head vs Multi-head attention

Single Head Attention
Our Case

3.864257	3.79246	4.060367	3.39751
3.801295	3.75252	3.977937	3.30861
3.855542	3.787426	4.04909	3.385086
3.622841	3.584936	3.750419	3.081834
3.745786	3.706744	3.904894	3.233519
3.835366	3.77523	4.022837	3.356435



Multi Head Attention (N Heads)
Real world Case Concatenation



Single Head attention vs Multi Head attention

Step 7. Multi Head Attention:
Normalizing single-head attention

$\text{softmax}\left(\frac{Q.K^T}{\sqrt{d_k}}\right)$ X Value

3.86	3.79	4.06	3.4
3.8	3.75	3.98	3.31
3.86	3.79	4.05	3.39
3.62	3.58	3.75	3.08
3.75	3.71	3.9	3.23
3.84	3.78	4.02	3.36

6 x 4

X

Linear weights
columns length must be
(embedding+positional) matrix columns length

0.8	0.34	0.45	0.54	0.07	0.53
0.85	0.74	0.78	0.5	0.75	0.55
0.53	0.81	0.55	0.59	0.49	0.14
0.7	0.6	0.12	0.42	0.29	0.87

4 x 6

=

10.84	9.45	7.33	7.8	6.09	7.66
10.65	9.28	7.22	7.67	5.99	7.51
10.83	9.43	7.33	7.79	6.08	7.65
10.08	8.77	6.85	7.25	5.67	7.09
10.48	9.12	7.11	7.54	5.89	7.38
10.77	9.38	7.29	7.75	6.05	7.6

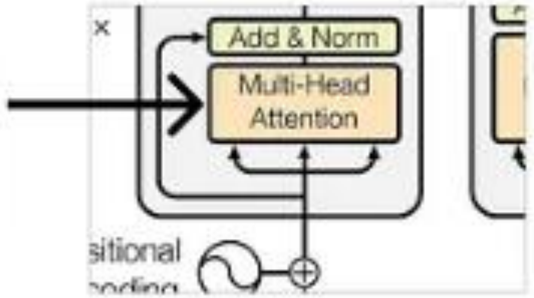
normalizing single head attention matrix

Step 7. Multi Head Attention:

Output of Multi Head attention

10.84	9.45	7.33	7.8	6.09	7.66
10.65	9.28	7.22	7.67	5.99	7.51
10.83	9.43	7.33	7.79	6.08	7.65
10.08	8.77	6.85	7.25	5.67	7.09
10.48	9.12	7.11	7.54	5.89	7.38
10.77	9.38	7.29	7.75	6.05	7.6

6 x 6



Output matrix of multi head attention

Step 8. Adding and Normalizing:

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6 x 6

+

Output of Multi Head attention

10.84	9.45	7.33	7.8	6.09	7.66
10.65	9.28	7.22	7.67	5.99	7.51
10.83	9.43	7.33	7.79	6.08	7.65
10.08	8.77	6.85	7.25	5.67	7.09
10.48	9.12	7.11	7.54	5.89	7.38
10.77	9.38	7.29	7.75	6.05	7.6

6 x 6

=

11.63	11.05	8.29	9.44	7.06	8.86
11.87	9.45	7.28	8.46	6.89	8.25
11.75	10.94	7.6	9.1	6.64	9.24
10.34	9.51	7.51	7.47	5.74	7.46
10.6	9.71	7.91	8.16	6.39	8.08
10.41	10.68	8.05	9.23	6.99	8.81

Adding matrices to perform add and norm step

Step 8. Adding and Normalizing:

$$mean = \frac{\sum_{i=1}^{N} X_i}{N} \qquad standard\ dev. = \sqrt{\frac{\sum_{i=1}^{N} (X_i - \mu)^2}{N}}$$

Row Wise Implementation

						Mean	Standard Deviation
11.63	11.05	8.29	9.44	7.06	8.86	9.26	1.57
11.87	9.45	7.28	8.46	6.89	8.25	8.56	1.64
11.75	10.94	7.6	9.1	6.64	9.24	9.04	1.76
10.34	9.51	7.51	7.47	5.74	7.46	7.86	1.51
10.6	9.71	7.91	8.16	6.39	8.08	8.37	1.35
10.41	10.68	8.05	9.23	6.99	8.81	8.93	1.28

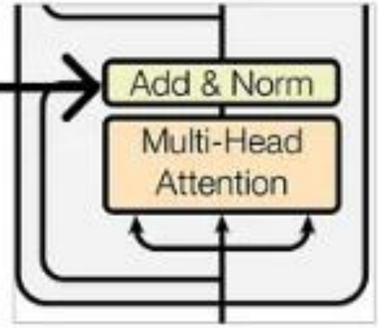
calculating meand and std.

Step 8. Adding and Normalizing:

11.63	11.05	8.29	9.44	7.06	8.86
11.87	9.45	7.28	8.46	6.89	8.25
11.75	10.94	7.6	9.1	6.64	9.24
10.34	9.51	7.51	7.47	5.74	7.46
10.6	9.71	7.91	8.16	6.39	8.08
10.41	10.68	8.05	9.23	6.99	8.81

Mean	Std
9.26	1.57
8.56	1.64
9.04	1.76
7.86	1.51
8.37	1.35
8.93	1.28

$$\frac{\text{value} - \text{mean}}{\text{std} + \text{error}} = \frac{11.63 - 9.26}{1.57 + 0.0001}$$



=

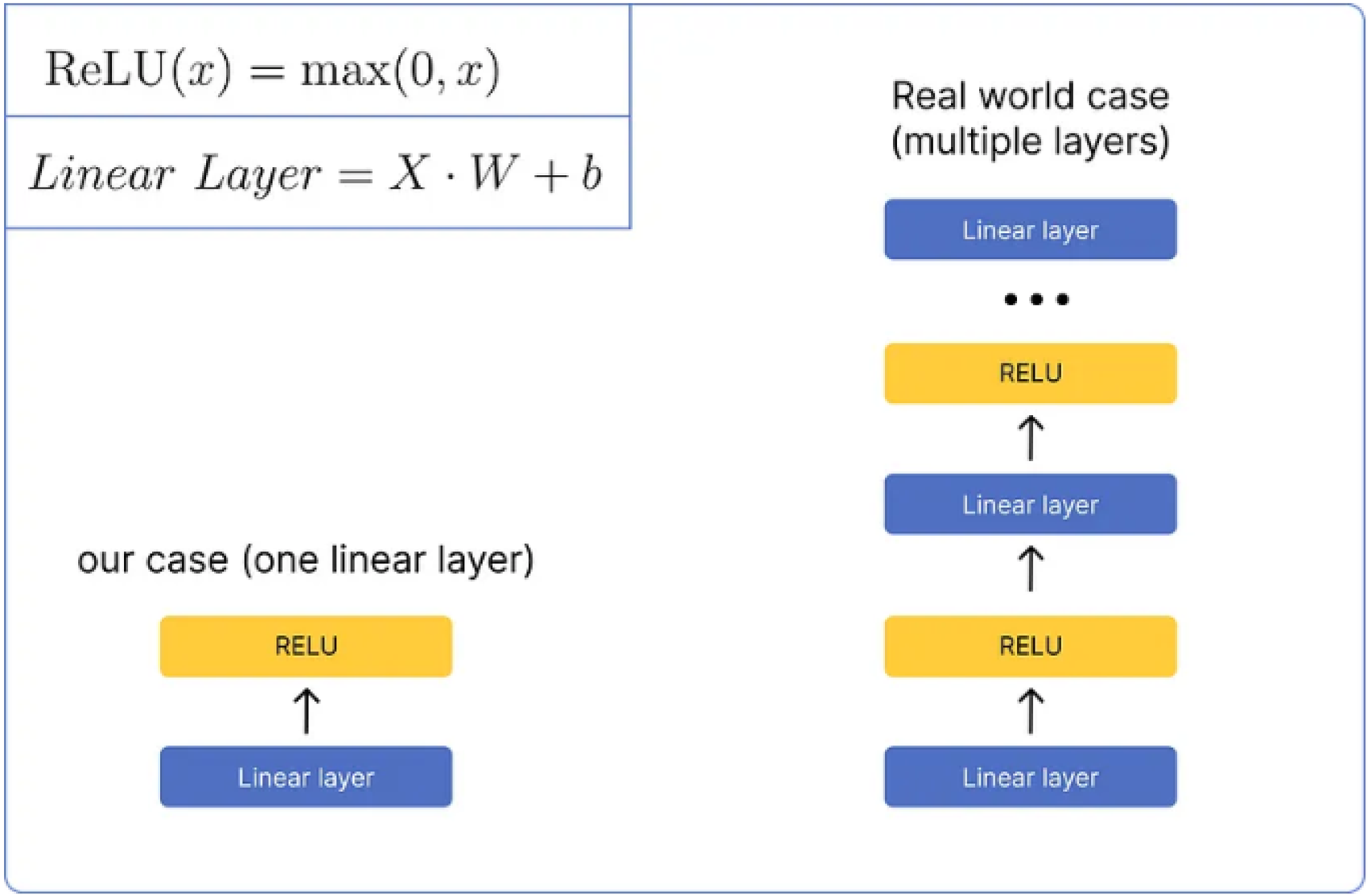
1.51	1.14	-0.62	0.11	-1.4	-0.25
2.02	0.54	-0.78	-0.06	-1.02	-0.19
1.54	1.08	-0.82	0.03	-1.36	0.11
1.64	1.09	-0.23	-0.26	-1.4	-0.26
1.65	0.99	-0.34	-0.16	-1.47	-0.21
1.16	1.37	-0.69	0.23	-1.52	-0.09

normalizing the resultant matrix

Step 9. Feed Forward Network:

$$\mathrm{ReLU}(x) = \max(0, x)$$

$$Linear\ Layer = X \cdot W + b$$



Feed Forward network comparison

Step 9. Feed Forward Network:

Matrix afrer add and norm step

1.51	1.14	-0.62	0.11	-1.4	-0.25
2.02	0.54	-0.78	-0.06	-1.02	-0.19
1.54	1.08	-0.82	0.03	-1.36	0.11
1.64	1.09	-0.23	-0.26	-1.4	-0.26
1.65	0.99	-0.34	-0.16	-1.47	-0.21
1.16	1.37	-0.69	0.23	-1.52	-0.09

6 x 6

X

W

0.5	0.05	0.97	0.22	0.56	0.02
0.17	0.52	0.63	0.48	0.06	0.6
0.53	0.87	0.47	0.1	0.31	0.79
0.83	0.58	0.38	0.09	0.64	0.25
0.81	0.85	0.74	0.35	0.31	0.53
0.25	0.31	0.22	0.77	0.57	0.85

6 x 6

$X \cdot W$

0.49	1.07	0.84	0.14	0.22	0.7
0.24	1.26	1.11	0.12	0.46	0.97
0.53	1.18	-0.82	0.39	0.33	0.59
0.53	0.97	0.98	0.15	0.16	0.52
0.56	1.11	-0.87	0.11	0.2	0.64
0.62	1.02	0.61	0.26	0.14	0.52

6 x 6

+

Bias

b1	b2	b3	b4	b5	b6
0.42	0.18	0.25	0.42	0.35	0.45

=

0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	-0.57	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	-0.62	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97

6 x 6

Calculating Linear Layer

Step 9. Feed Forward Network:

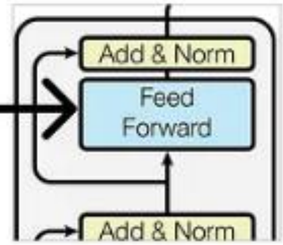
$$\mathrm{ReLU}(x) = \max(0, x)$$

0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	-0.57	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	-0.62	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97

6 x 6

$\max(0, 0.91)$

0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	0	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	0	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97



Calculating ReLU Layer

Step 10. Adding and Normalizing Again:

Matrix from Feed Forward Network

0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	0	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	0	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97

+

Matrix from Previous Add and Norm Step

1.51	1.14	-0.62	0.11	-1.4	-0.25
2.02	0.54	-0.78	-0.06	-1.02	-0.19
1.54	1.08	-0.82	0.03	-1.36	0.11
1.64	1.09	-0.23	-0.26	-1.4	-0.26
1.65	0.99	-0.34	-0.16	-1.47	-0.21
1.16	1.37	-0.69	0.23	-1.52	-0.09

=

2.42	2.39	0.47	0.67	-0.83	0.9
2.68	1.98	0.58	0.48	-0.21	1.23
2.49	2.44	-0.82	0.84	-0.68	1.15
2.59	2.24	1	0.31	-0.89	0.71
2.63	2.28	-0.34	0.37	-0.92	0.88
2.2	2.57	0.17	0.91	-1.03	0.88

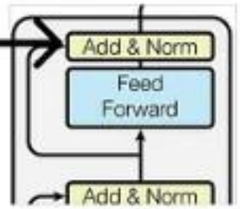
→

Mean	Std
1.0033	1.103534
1.1233	1.214349
0.9033	1.301837
0.9933	1.289055
0.8167	1.306016
0.95	1.320773

=

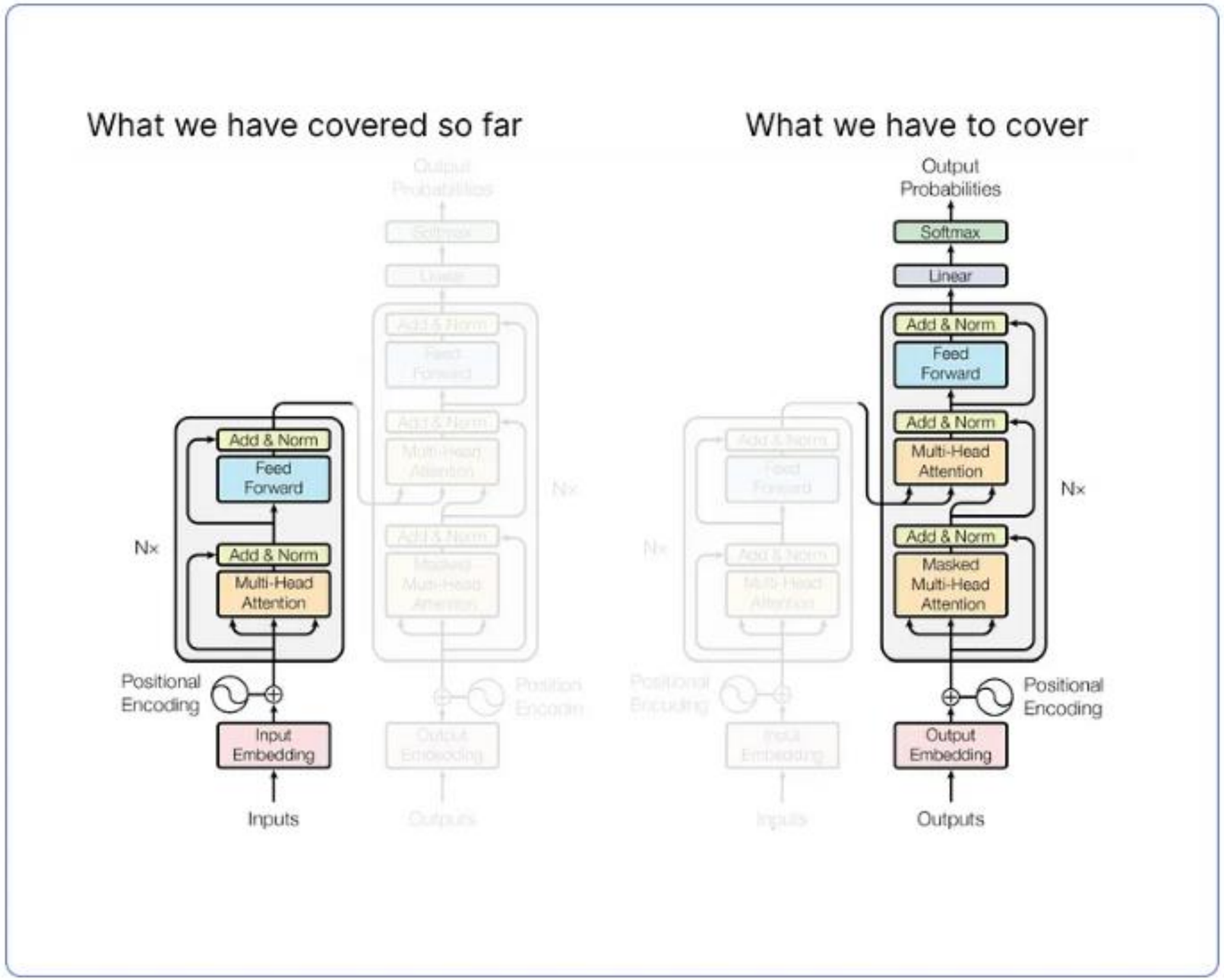
1.28	1.26	-0.48	-0.3	-1.66	-0.09
1.28	0.71	-0.45	-0.53	-1.1	0.09
1.22	1.18	-1.32	-0.05	-1.22	0.19
1.24	0.97	0.01	-0.53	-1.46	-0.22
1.39	1.12	-0.89	-0.34	-1.33	0.05
0.95	1.23	-0.59	-0.03	-1.5	-0.05

→

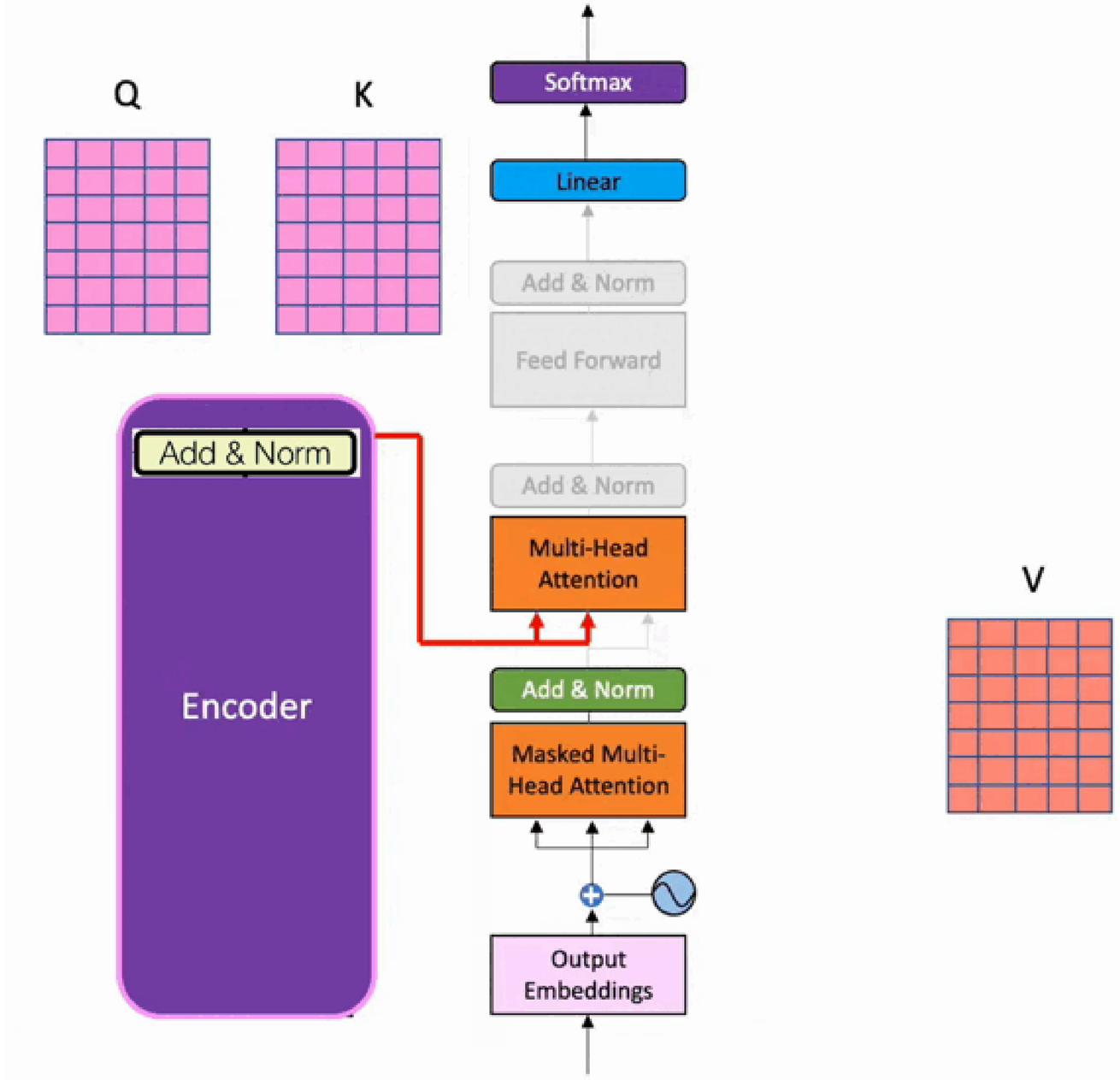


Add and Norm after Feed Forward Network

Step 11. Decoder Part:

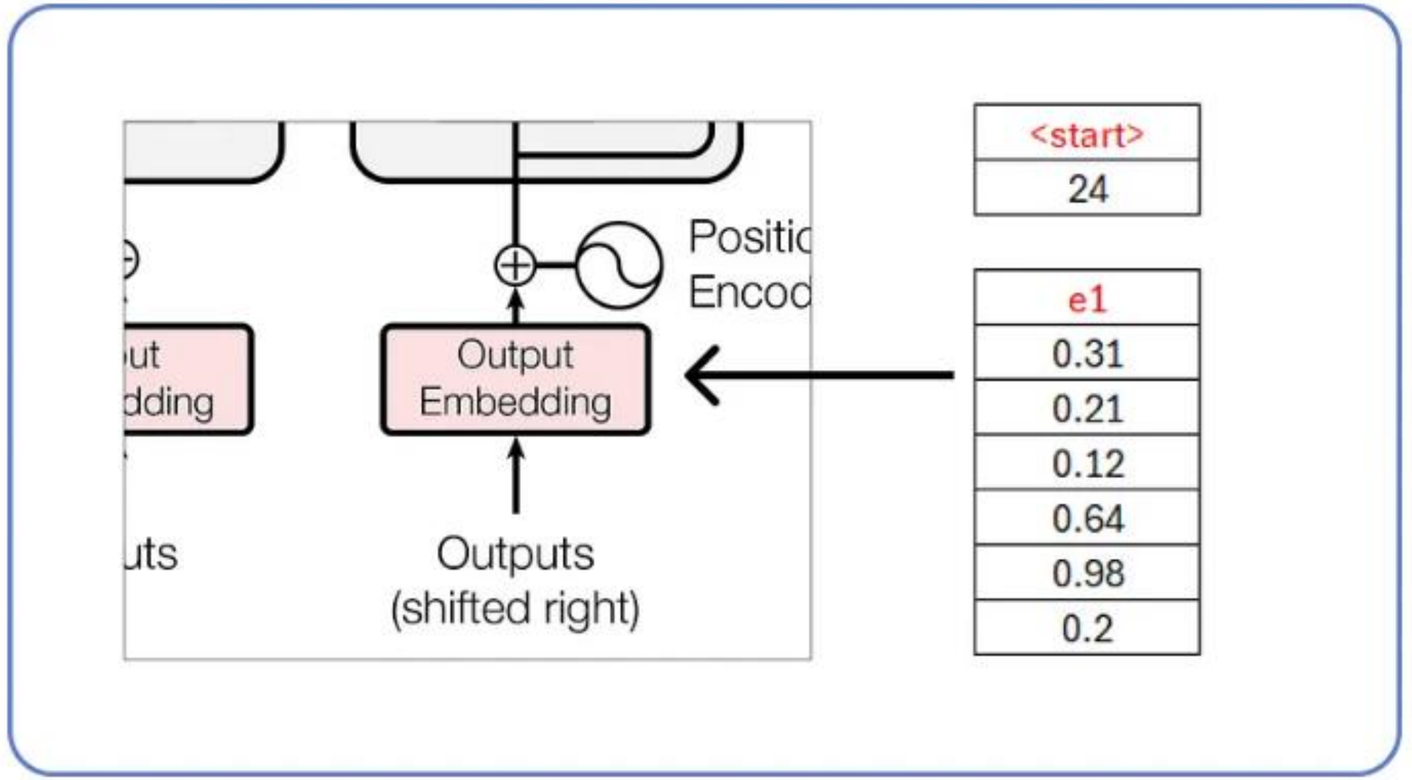


Step 11. Decoder Part:



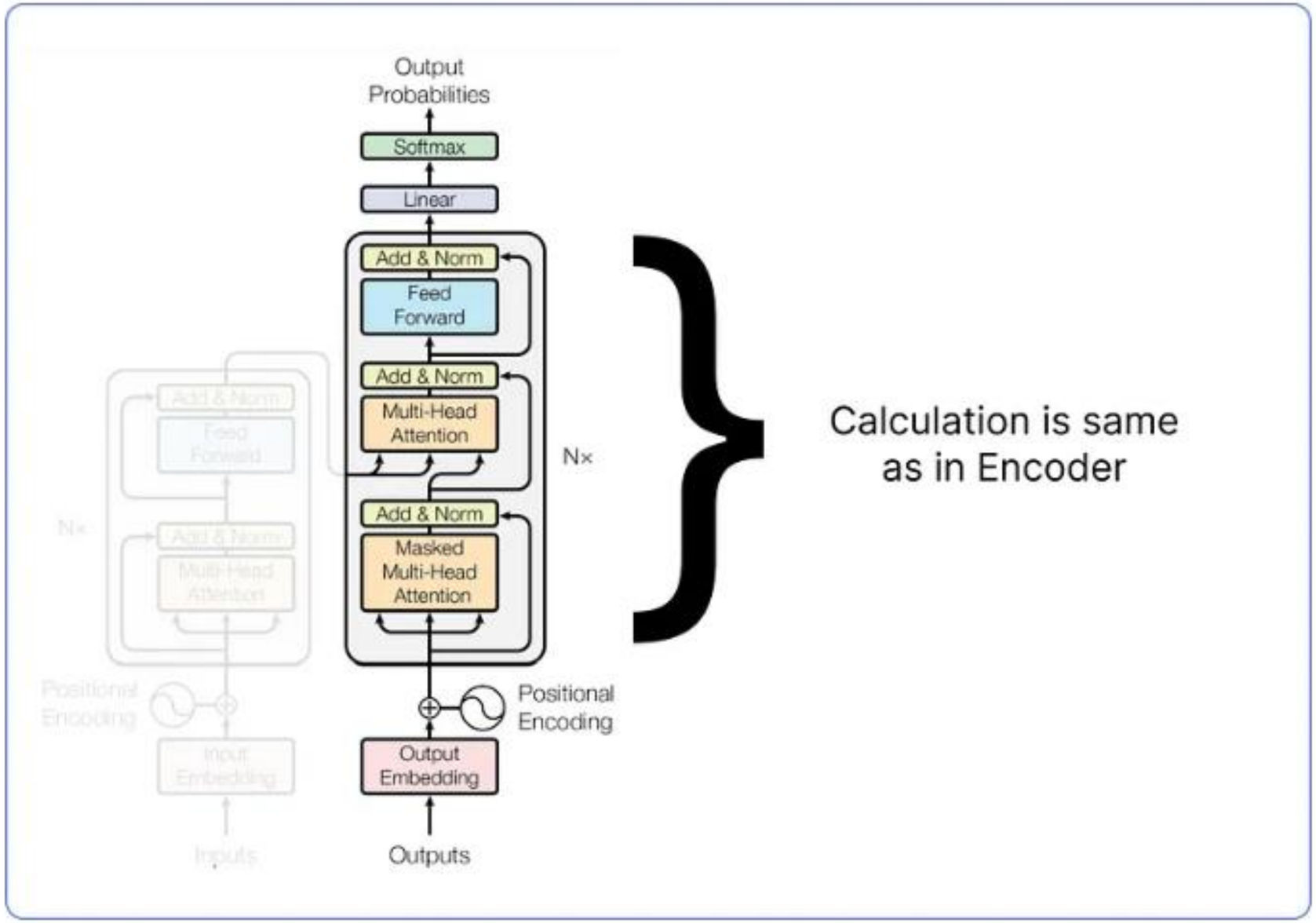
Step 11. Decoder Part:

input comparison of encoder and decoder



Decoder input **<start>** word

Step 11. Decoder Part:



Calculating Decoder

Step 12. Understanding Mask Multi Head Attention

$$\text{Input Matrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

inpur matrix for masked multi head attentions

Linear Projections (Query, Key, Value): Assume the linear projections for each head:

Head 1: *Wq*1,*Wk*1,*Wv*1 and **Head 2: *Wq*2,*Wk*2,*Wv*2**

Calculate Attention Scores: For each head, calculate attention scores using the dot product of Query and Key, and apply the mask to prevent attending to future positions.

Apply Softmax: Apply the softmax function to obtain attention weights.

Weighted Summation (Value): Multiply the attention weights by the Value to get the weighted sum for each head.

Concatenate and Linear Transformation: Concatenate the outputs from both heads and apply a linear transformation.

Step 12. Understanding Mask Multi Head Attention

$$\text{Input Matrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

inpur matrix for masked multi head attentions

- ***Wq*1 = *Wk*1 = *Wv*1 = *Wq*2 = *Wk*2 = *Wv*2 = *I***, the identity matrix.
- ***Q*=*K*=*V*=Input Matrix**

Step 12. Understanding Mask Multi Head Attention

Head 1:

$$Q_1 = K_1 = V_1 = \text{Input Matrix}$$

$$A_1 = Q_1 \cdot K_1^T$$

$$A_1 = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

$$A_1 = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 5 & 0 \\ 7 & 8 & 9 \end{bmatrix} \quad \text{(Masked)}$$

$$W_1 = \text{softmax}(A_1)$$

$$O_1 = W_1 \cdot V_1$$

Head 2:

$$Q_2 = K_2 = V_2 = \text{Input Matrix}$$

$$A_2 = Q_2 \cdot K_2^T$$

$$A_2 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 0 & 0 & 9 \end{bmatrix} \quad \text{(Masked)}$$

$$W_2 = \text{softmax}(A_2)$$

$$O_2 = W_2 \cdot V_2$$

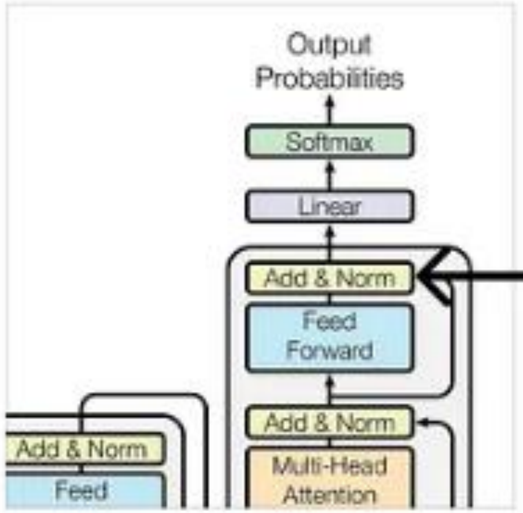
Concatenate and Linear Transformation:

$$\text{Concatenate}([O_1, O_2])$$

$$\text{(Apply Learnable Linear Transformation)}$$

Mask Multi Head Attention (**Two Heads**)

Step 13. Calculating the Predicted Word

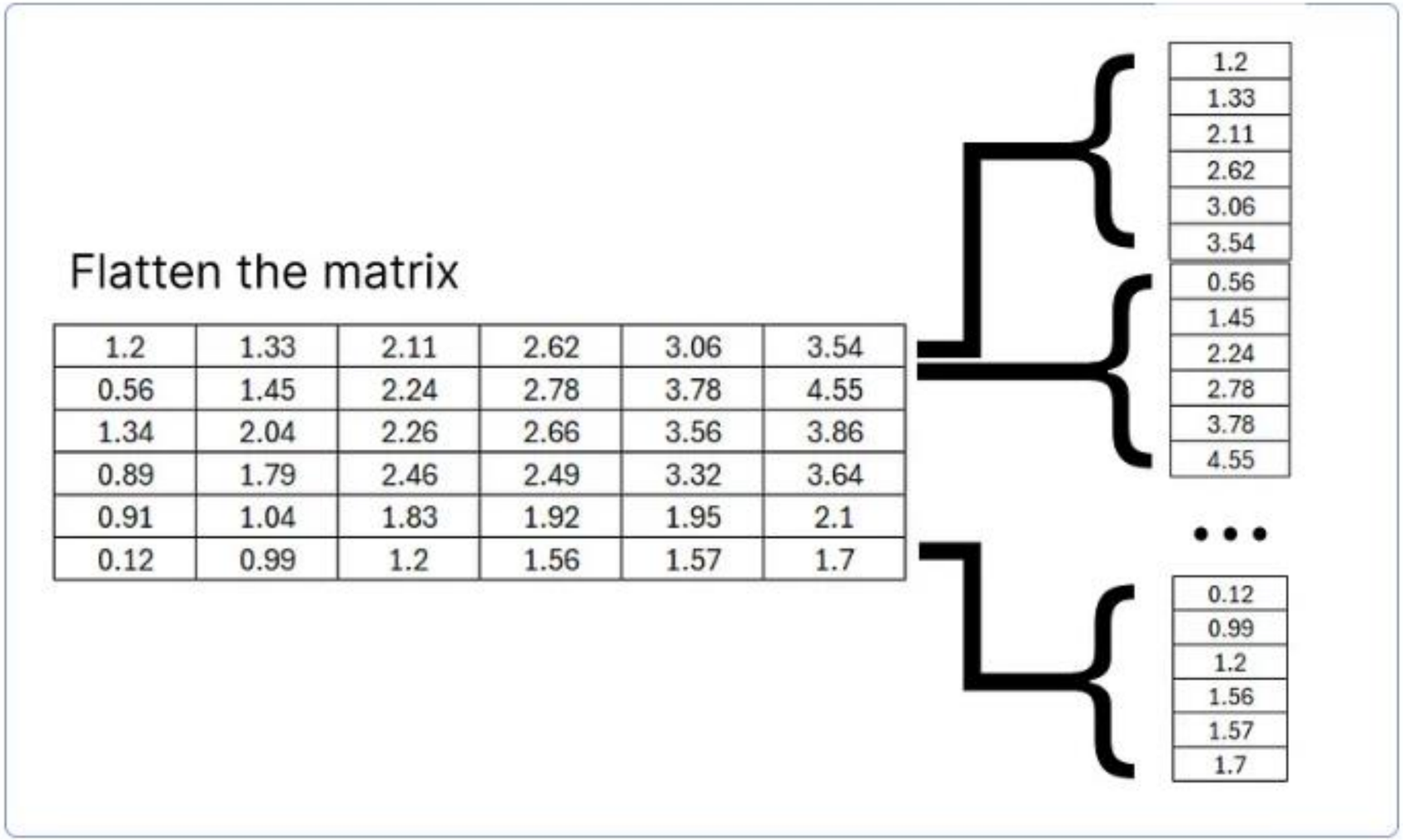


Rows must be 6 while columns can be any length

1.2	1.33	2.11	2.62	3.06	3.54
0.56	1.45	2.24	2.78	3.78	4.55
1.34	2.04	2.26	2.66	3.56	3.86
0.89	1.79	2.46	2.49	3.32	3.64
0.91	1.04	1.83	1.92	1.95	2.1
0.12	0.99	1.2	1.56	1.57	1.7

Add and Norm output of decoder

Step 13. Calculating the Predicted Word



flattened the last add and norm block matrix

Step 13. Calculating the Predicted Word

$$Linear\ Layer = X \cdot W$$

No Bias

Linear set of weights

I	drink	things	...	He
1	2	3	...	23
0.28	0.72	0.14		0.11
0.1	0.5	0.93		0.37
0.77	0.23	0.17	...	0.51
0.23	0.62	0.99		0.76
0.75	0.76	0.42	...	0.26
0.02	0.52	0.47		0.24
0.58	0.9	0.88		0.54
...	...	...	...	...
0.59	0.16	0.05		0.87
0.35	0.02	0.84		0.67

Flatten Layer (Row Matrix)

1.2	1.33	2.11	...	1.2	1.56	1.57	1.7

1 x n (our case n is 36)

X

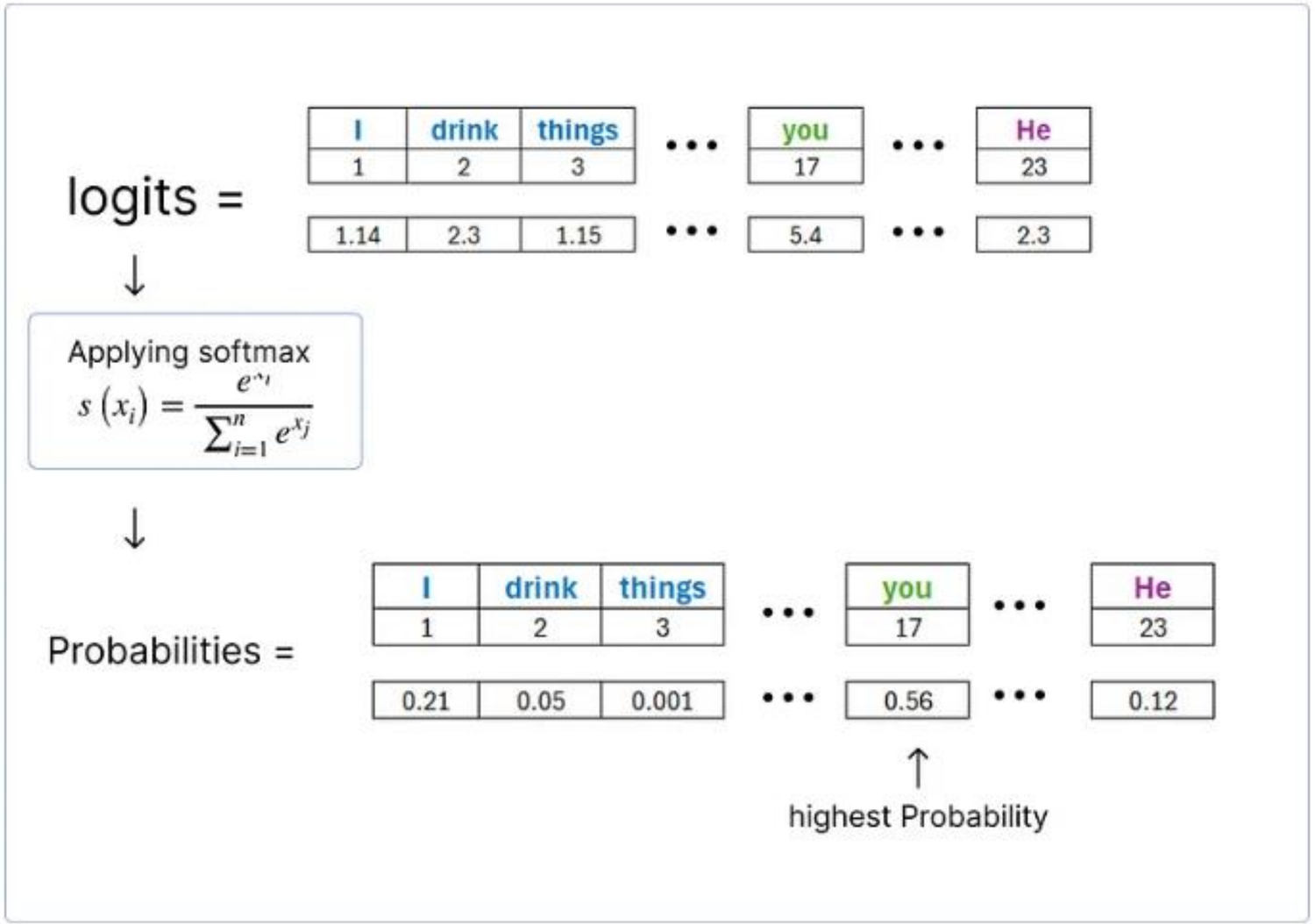
n x m
(m is 23 Vocab Size)

logits =

I	drink	things	...	you	...	He
1	2	3		17		23
1.14	2.3	1.15	...	5.4	...	2.3

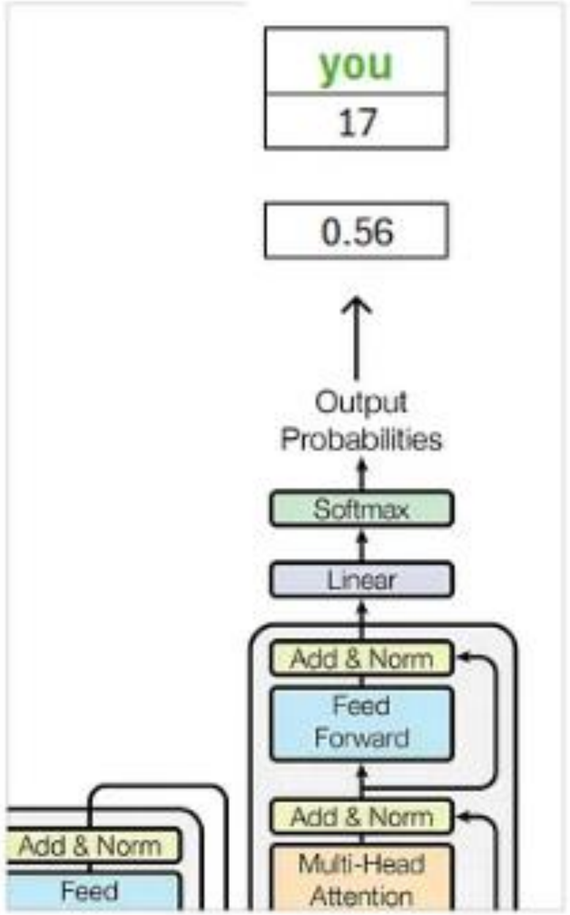
Calculating Logits

Step 13. Calculating the Predicted Word



Finding the Predicted word

Step 13. Calculating the Predicted Word

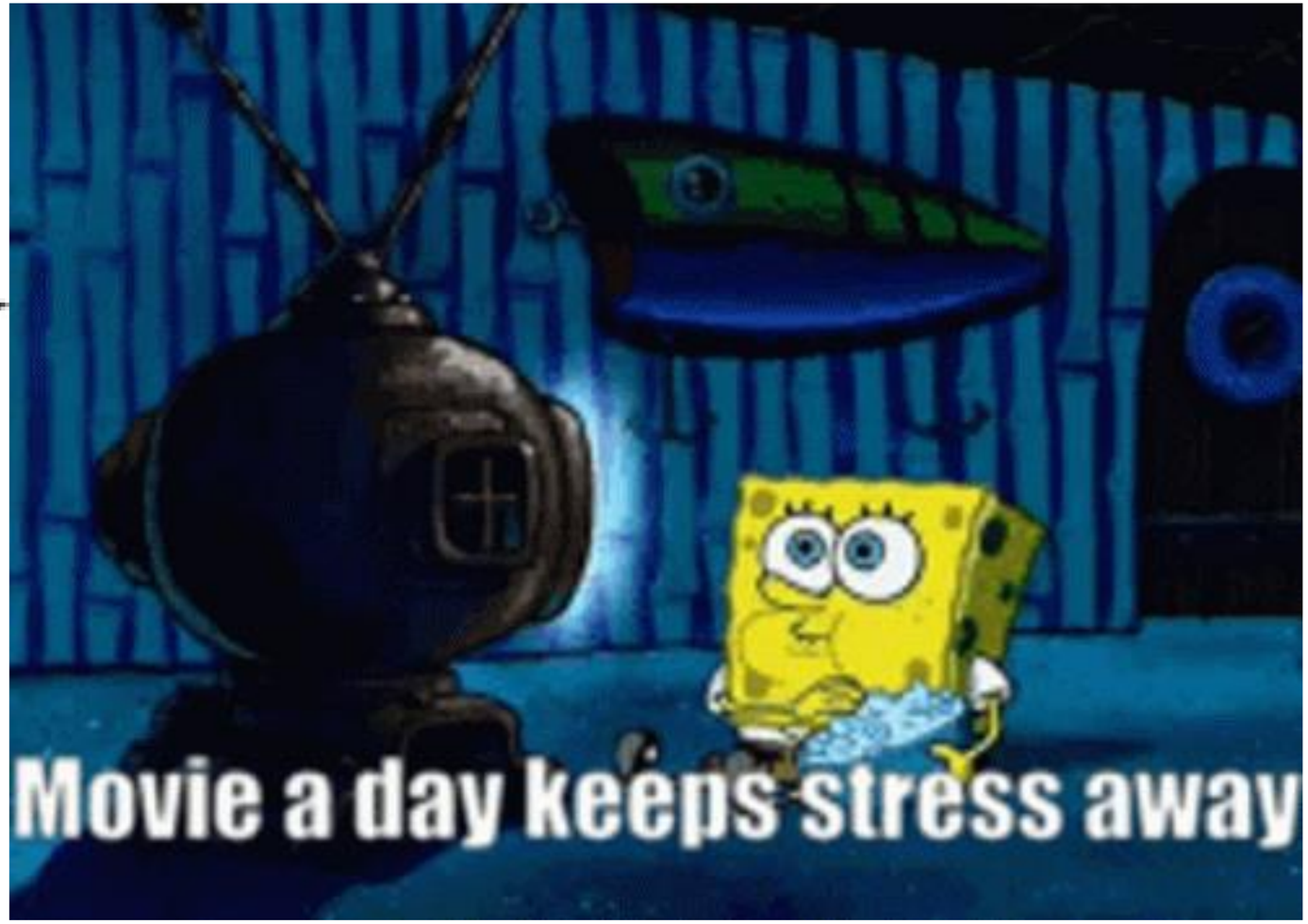


you is the predicted word, which will now act an an input for our decoder and so on...

Final output of decoder

WHAT IS ATTENTION?


Movie
Time!


Movie a day keeps stress away

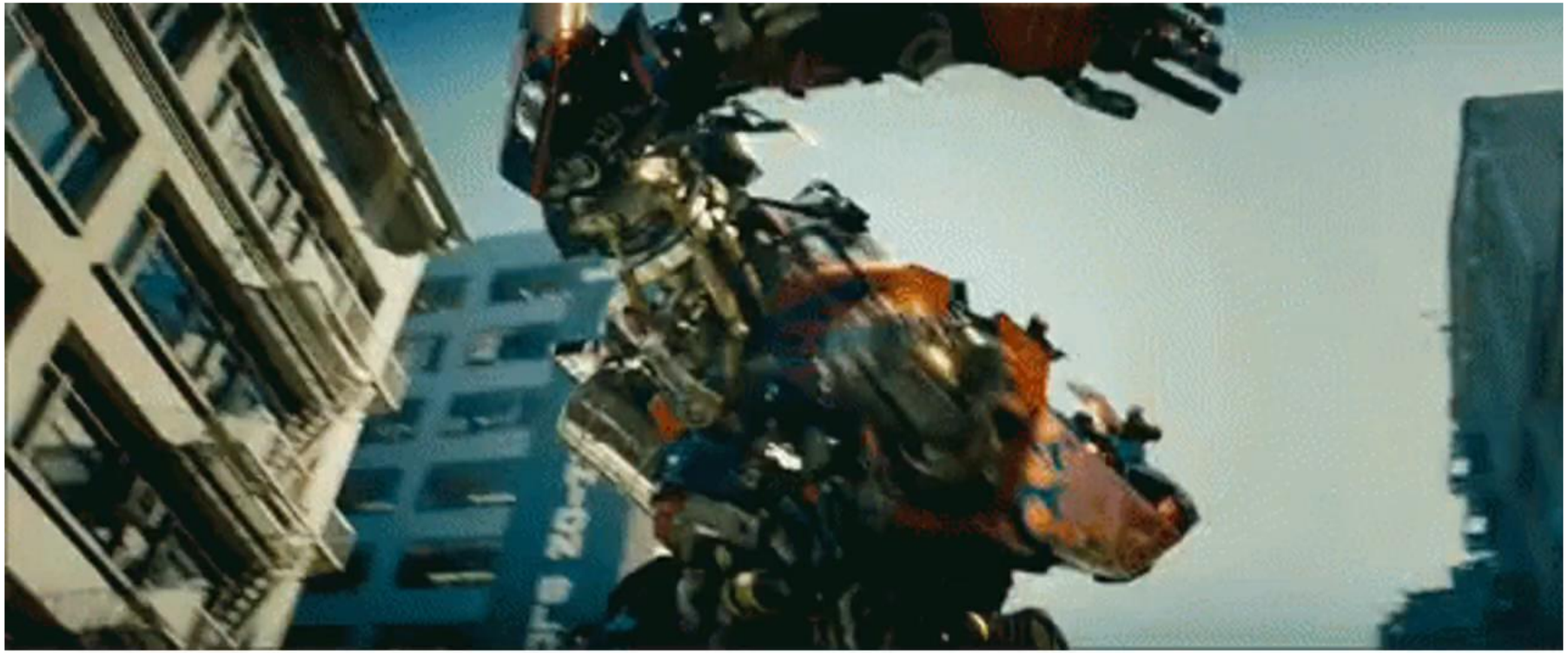
ATTENTION

Source: https://www.youtube.com/watch?v=48gBPL7aHJY&t=1s

POSITION EMBEDDINGS

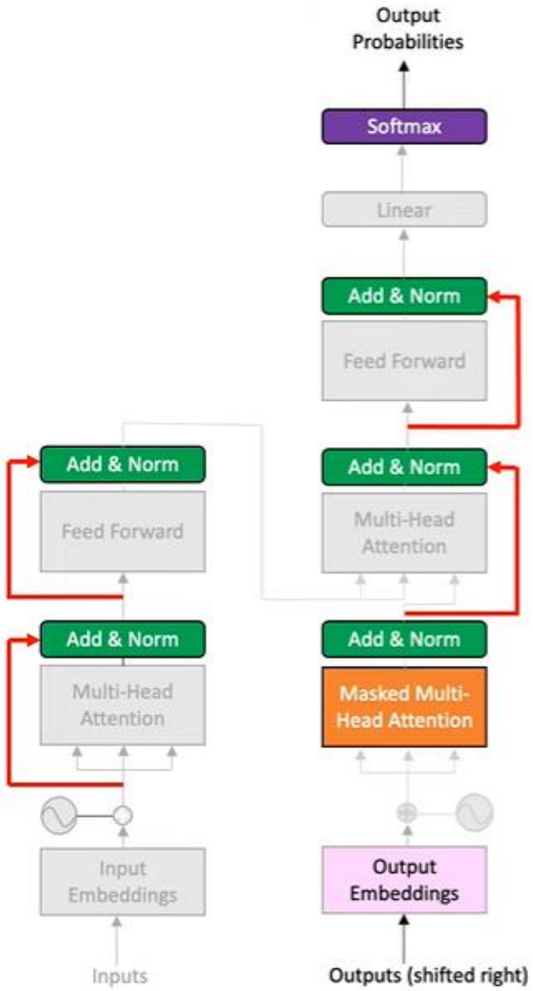
Source: https://www.youtube.com/watch?v=dichIcUZfOw&t=65s

MULTI-HEAD & SELF-ATTENTION



Source: https://www.youtube.com/watch?v=mMa2PmYJlCo

DECODER'S MASKED ATTENTION



Episode 3

1. Residual Connections
2. Layer Normalization
3. Decoder
   i. Important components
   ii. Masked-Attention

Source: https://www.youtube.com/watch?v=gJ9kaJsE78k

Building Transformer from Scratch in PyTorch

Installing and Upgrading Torch Transformer with pip

```
!pip install --upgrade torch transformers
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (2.0.1+cu118)
Collecting transformers
  Downloading transformers-4.29.2-py3-none-any.whl (7.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.1/7.1 MB 52.1 MB/s eta 0:00:00
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch) (3.12.0)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from torch) (4.5.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch) (1.11.1)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch) (3.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch) (3.1.2)
Requirement already satisfied: triton==2.0.0 in /usr/local/lib/python3.10/dist-packages (from torch) (2.0.0)
Requirement already satisfied: cmake in /usr/local/lib/python3.10/dist-packages (from triton==2.0.0->torch) (3.25.2)
Requirement already satisfied: lit in /usr/local/lib/python3.10/dist-packages (from triton==2.0.0->torch) (16.0.5)
Collecting huggingface-hub<1.0,>=0.14.1 (from transformers)
  Downloading huggingface_hub-0.14.1-py3-none-any.whl (224 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 224.5/224.5 kB 19.7 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-packages (from transformers) (1.22.4)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from transformers) (23.1)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-packages (from transformers) (6.0)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (from transformers) (2022.10.31)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from transformers) (2.27.1)
Collecting tokenizers!=0.11.3,<0.14,>=0.11.1 (from transformers)
  Downloading tokenizers-0.13.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (7.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.8/7.8 MB 100.3 MB/s eta 0:00:00
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/dist-packages (from transformers) (4.65.0)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.14.1->transformers) (2023.4.0
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch) (2.1.2)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (1.26.15)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (2022.12.7)
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (2.0.12)
```

Building Transformer Model from Scratch

```
# We first import all the necessary libraries. torch is the main PyTorch library,
# nn is the Neural Networks library, filled with various layers and loss functions.
import torch
from torch import nn
from torch.nn import CrossEntropyLoss
from torch.optim import Adam
from torch.utils.data import DataLoader, Dataset
import math
import torch.nn.functional as F

# transformers library provides pre-trained models and tokenizers.
from transformers import BertTokenizer

# We create an instance of the BERT tokenizer.
# This tokenizer will convert our text data into tokens which the BERT model can understand.
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

We use BERT tokenizer from transformers library.

Building Transformer Model from Scratch

```
# Define the Multi-head Attention mechanism as used in the Transformer model. This class inherits from torch.nn.Module.
class MultiHeadAttention(nn.Module):
    """
    * n_heads: This parameter refers to the number of parallel attention layers (or "heads") in the MultiHeadAttention mechanism.
               These heads allow the model to focus on different parts of the input for each head, capturing various aspects of the data.
    * d_model: This parameter represents the total dimension of the model or the input embedding.
    * d_k and d_v: These parameters are the dimensions of the keys and values used in the MultiHeadAttention mechanism.
                   Usually d_k and d_v are the same.
    """
    def __init__(self, n_heads, d_model, d_k, d_v, dropout=0.1):
        super().__init__()
        self.n_heads = n_heads
        self.d_k = d_k
        self.d_v = d_v
        """
        *  self.w_q, self.w_k, self.w_v, self.w_o: These are the weight matrices that the model learns to effectively perform the attention mechanism.
        *  w_q, w_k, and w_v transform the input into query, key, and value respectively for all heads.
        *  w_o is applied to the concatenated output of the attention heads.
        """
        # The following are linear layers used for transforming the input and output.
        self.w_q = nn.Linear(d_model, d_k * n_heads, bias=False)
        self.w_k = nn.Linear(d_model, d_k * n_heads, bias=False)
        self.w_v = nn.Linear(d_model, d_v * n_heads, bias=False)
        self.w_o = nn.Linear(n_heads * d_v, d_model, bias=False)
        """
        dropout: This parameter controls the rate at which randomly selected neurons in
        a layer are ignored (or "dropped out") during training. This helps to prevent overfitting.
        """
        self.dropout = nn.Dropout(dropout)
```

Building Transformer Model from Scratch

```
def forward(self, q, k, v, mask=None):
    # The forward method defines the operations performed on the input data.
    batch_size, seq_length, _ = q.size()

    # Apply the linear layers to the inputs
    q = self.w_q(q).view(batch_size, seq_length, self.n_heads, self.d_k)
    k = self.w_k(k).view(batch_size, seq_length, self.n_heads, self.d_k)
    v = self.w_v(v).view(batch_size, seq_length, self.n_heads, self.d_v)

    # Transpose the dimensions for matrix multiplication
    q = q.transpose(1, 2)
    k = k.transpose(1, 2)
    v = v.transpose(1, 2)

    # Compute scaled dot-product attention scores
    scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(self.d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)  # Apply mask
    scores = F.softmax(scores, dim=-1)  # Apply softmax to the scores
    scores = self.dropout(scores)  # Apply dropout to the scores
    output = torch.matmul(scores, v)  # Compute the output

    # Reshape and apply the output linear layer
    output = output.transpose(1, 2).contiguous().view(batch_size, -1, self.n_heads * self.d_v)
    output = self.w_o(output)
    return output
```

Building Transformer Model from Scratch

```python
# Define a class for the Positionwise Feed-Forward Network, which is another key component of the Transformer model.
class PositionwiseFeedForward(nn.Module):
    """
    * d_ff: This parameter is the dimensionality of the "inner" feed-forward layer of the transformer.

    """
    def __init__(self, d_model, d_ff, dropout=0.1):
        super().__init__()
        self.w_1 = nn.Linear(d_model, d_ff)  # First linear layer
        self.w_2 = nn.Linear(d_ff, d_model)  # Second linear layer
        self.dropout = nn.Dropout(dropout)  # Dropout layer

    def forward(self, x):
        return self.w_2(self.dropout(F.relu(self.w_1(x))))  # Return the output of the Feed-Forward Network
```

Building Transformer Model from Scratch

```python
# Define a class for a Transformer Block, which consists of one Multi-Head Attention layer and one Position-wise Feed-Forward Network.
class TransformerBlock(nn.Module):
    def __init__(self, d_model, d_ff, d_k, d_v, n_heads, dropout=0.1):
        super().__init__()
        self.attn = MultiHeadAttention(n_heads, d_model, d_k, d_v, dropout)  # Multi-head attention layer
        self.ffn = PositionwiseFeedForward(d_model, d_ff, dropout)  # Feed-forward layer
        self.lnorm_1 = nn.LayerNorm(d_model)  # Layer normalization
        self.lnorm_2 = nn.LayerNorm(d_model)  # Layer normalization
        self.dropout_1 = nn.Dropout(dropout)  # Dropout layer
        self.dropout_2 = nn.Dropout(dropout)  # Dropout layer

    def forward(self, x, mask=None):
        _x = x
        x = self.attn(x, x, x, mask)
        x = self.lnorm_1(_x + self.dropout_1(x))
        _x = x
        x = self.ffn(x)
        x = self.lnorm_2(_x + self.dropout_2(x))
        return x
```

Building Transformer Model from Scratch

```python
# Define the complete Transformer model
class TransformerModel(nn.Module):
    def __init__(self, d_model, d_ff, d_k, d_v, n_heads, vocab_size, dropout=0.1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)  # Embedding layer
        self.transformer_block = TransformerBlock(d_model, d_ff, d_k, d_v, n_heads, dropout)  # Transformer block
        self.output_layer = nn.Linear(d_model, vocab_size)  # Output layer

    def forward(self, x, mask=None):
        x = self.embedding(x)  # Apply embedding
        x = self.transformer_block(x, mask)
        x = self.output_layer(x)
        return x
```

Building Transformer Model from Scratch

```
# Define a class for handling the dataset.
class TextDataset(Dataset):
    def __init__(self, text, block_size):
        self.data = tokenizer.encode(text)  # Encode the input text
        self.block_size = block_size

    def __len__(self):
        return max(0, len(self.data) - self.block_size)  # Length of the data

    def __getitem__(self, idx):
        chunk = self.data[idx:idx + self.block_size + 1]
        dix = chunk[:-1]
        yix = chunk[1:]
        x = torch.tensor(dix, dtype=torch.long)
        y = torch.tensor(yix, dtype=torch.long)
        return x, y

# Initialize the dataset and the DataLoader
"""
* block_size: This is the size of the context window that is fed into the model.
In language tasks, this would be the number of words or characters that the model looks at for each step.
"""
block_size = 16
text = "Tell me and I forget. Teach me and I remember. Involve me and I learn."
dataset = TextDataset(text, block_size)
"""
* batch_size: This parameter specifies the number of samples to pass through the model at once.
It affects the speed and memory usage of the training process.
"""
dataloader = DataLoader(dataset, batch_size=2, shuffle=True)
```

Building Transformer Model from Scratch

```
# Initialize the model, loss function, and optimizer
"""
* vocab_size: This parameter refers to the size of the vocabulary.
It's the number of unique words or characters that the model can possibly predict.
"""
vocab_size = len(tokenizer.get_vocab())
model = TransformerModel(d_model=64, d_ff=256, d_k=64, d_v=64, n_heads=8, vocab_size=vocab_size, dropout=0.1)
criterion = CrossEntropyLoss()  # Cross-entropy loss for classification tasks
"""
* lr (Learning rate): This parameter controls the size of the adjustments
  to the weights that the backpropagation step makes.
"""
optimizer = Adam(model.parameters(), lr=1e-3)  # Adam optimizer with learning rate 1e-3.

# Start the training loop
model.train()
for epoch in range(10):
    for x, y in dataloader:
        optimizer.zero_grad()  # Reset gradients
        output = model(x)  # Forward pass
        output = output.view(-1, vocab_size)
        y = y.view(-1)
        loss = criterion(output, y)  # Compute the loss
        loss.backward()  # Backpropagation
        optimizer.step()  # Update the weights
        print(f"Epoch: {epoch}, Loss: {loss.item()}")  # Print the loss for this epoch
```

```
Epoch: 0, Loss: 10.296504020690918
Epoch: 0, Loss: 9.885079383850098
Epoch: 1, Loss: 9.552763938903809
Epoch: 1, Loss: 9.109817504882812
Epoch: 2, Loss: 8.877806663513184
Epoch: 2, Loss: 8.60193920135498
Epoch: 3, Loss: 8.396543502807617
Epoch: 3, Loss: 8.141138076782227
Epoch: 4, Loss: 7.931668281555176
Epoch: 4, Loss: 7.790417671203613
Epoch: 5, Loss: 7.573558807373047
Epoch: 5, Loss: 7.331594467163086
Epoch: 6, Loss: 7.155040740966797
Epoch: 6, Loss: 7.012888431549072
Epoch: 7, Loss: 6.8704729080200195
Epoch: 7, Loss: 6.582520484924316
Epoch: 8, Loss: 6.463815689086914
Epoch: 8, Loss: 6.391653537750244
Epoch: 9, Loss: 6.193269729614258
Epoch: 9, Loss: 6.0923333168029785
```

Evaluating the Performance of the Trained Model: Testing

```python
def predict_next_word(model, text, tokenizer):
    model.eval()

    # Prepare the inputs
    inputs = tokenizer.encode(text)
    inputs = torch.tensor(inputs, dtype=torch.long).unsqueeze(0)  # add batch dimension

    # Forward pass
    with torch.no_grad():
        output = model(inputs)

    # Get the last predicted token
    last_token_logits = output[0, -1, :]

    # Convert to probabilities (softmax)
    probs = F.softmax(last_token_logits, dim=-1)

    # Get the token id of the most probable token
    predicted_token_id = torch.argmax(probs).item()

    # Decode the token id to get the token
    predicted_token = tokenizer.decode([predicted_token_id])

    return predicted_token

text = "Tell me and I forget. Teach me and I remember. Involve me and I"
predicted_word = predict_next_word(model, text, tokenizer)
print(f"The predicted next word is: {predicted_word}")
```

```
The predicted next word is: forget
```

Building Tokenizer without using Pretrained BERT Tokenizer

Building Simple Tokenizer

```python
class SimpleTokenizer:
    def __init__(self, corpus):
        # Building the vocabulary from the corpus
        self.vocab = self.build_vocab(corpus)
        # Mapping from id to token
        self.id2token = {id: token for token, id in self.vocab.items()}

    def build_vocab(self, corpus):
        # Pre-defined tokens
        vocab = {'<pad>': 0, '<unk>': 1}
        words = corpus.lower().split()
        # Assigning an id to each unique word in the corpus
        for i, word in enumerate(set(words)):
            vocab[word] = i + 2
        return vocab

    def encode(self, text):
        # Encoding a text into tokens
        return [self.vocab.get(word, self.vocab['<unk>']) for word in text.lower().split()]

    def decode(self, token_ids):
        # Decoding tokens back to text
        return ' '.join(self.id2token.get(id, self.vocab['<unk>']) for id in token_ids)

    def get_vocab(self):
        # Returns the vocabulary
        return self.vocab
```

Utilizing Pretrained GPT-2 as a Plug-and-Play Transformer Architecture

Pretrained GPT-2: A Plug-and-Play Transformer Example

```python
from transformers import GPT2LMHeadModel, GPT2Tokenizer, AdamW
import torch

# Define the model and tokenizer
model_name = "gpt2"
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token  # Added padding token
model = GPT2LMHeadModel.from_pretrained(model_name)

# Text to train on
text = "Tell me and I forget. Teach me and I remember. Involve me and I learn."

# Tokenize the input text
inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True)

# Define the input and label tensors
input_ids = inputs["input_ids"]
labels = input_ids.detach().clone()

# Shift the labels tensor to the left and pad
labels[:-1] = input_ids[1:]
labels[-1] = -100

# Optimizer
optimizer = AdamW(model.parameters(), lr=1e-5)
```

Pretrained GPT-2: A Plug-and-Play Transformer Example

```
# Training loop
model.train()
for epoch in range(10):
    optimizer.zero_grad()
    outputs = model(input_ids, labels=labels)
    loss = outputs.loss
    loss.backward()
    optimizer.step()
    print(f"Epoch: {epoch}, Loss: {loss.item()}")

# Now you can generate text
generated = model.generate(input_ids, max_length=50, do_sample=True, temperature=0.7)
print(tokenizer.decode(generated[0]))
```

```
Tell me and I forget. Teach me and I remember. Involve me and I learn. Teach me and I teach me.

But I can't do it.

I can't do it.
```

Loss Function

Loss functions

- Loss functions are important tools in deep learning.
- They help us measure the difference between predicted outputs and actual targets.
- Loss functions guide the model's improvement by showing how well it's performing.

Purpose of Loss Functions

- **Optimization:** Loss functions help us make the model better by adjusting its parameters.
- **Error Measurement:** They tell us how much our model is getting things wrong.
- **Model Evaluation:** Loss functions allow us to compare different models and see which one performs better.
- **Regularization:** Some loss functions can prevent overfitting and make our model more generalizable.

Impact of Loss Functions

- **Training Dynamics:** The choice of loss function affects how the model learns and improves.
- **Model Performance:** The loss function directly affects the model's ability to make accurate predictions.
- **Optimization Guidance:** Loss functions provide directions for adjusting the model's parameters effectively.
- **Generalization:** Using the right loss function helps the model make accurate predictions on new data.

Pros and Cons

- **Mean Squared Error (MSE):** Simple, but sensitive to outliers.
- **Cross-Entropy Loss:** Good for multi-class classification, but may lead to large updates.
- **Binary Cross-Entropy Loss:** Suitable for binary classification and imbalanced classes.
- **Negative Log Likelihood Loss:** Ideal for multi-class classification, but requires log-probabilities.
- **L1 Loss:** Robust to outliers, but not differentiable at zero.

Choosing the Right Loss Function

- **Task Alignment:** Choose a loss function that matches the task and data characteristics.
- **Optimization and Generalization:** Use a loss function that minimizes the difference and improves generalization.
- **Model Comparison:** Compare different loss functions to find the best-performing model.
- **Deep Learning Success:** Selecting the right loss function is crucial for successful training.

Commonly Used Loss Functions

Mean Squared Error Loss (MSELoss)

Mean Squared Error (MSE) loss is commonly used in regression tasks. It calculates the squared differences between the target and the predicted values.

- **Pros:** It's simple and easy to understand. It punishes large errors more than smaller ones through the square term. It's differentiable, making it suitable for optimization.
- **Cons:** It can be sensitive to outliers because it squares the error. If your output isn't normally distributed, it might not be the best choice. It also assumes that errors are independent and identically distributed, which might not always be the case.
- **Common Usage:** Regression tasks.

```python
import torch
import torch.nn as nn

predicted = torch.randn(2, 2)
target = torch.randn(2, 2)
print("Predicted: ", predicted)
print("Target: ", target)

criterion = nn.MSELoss()

loss = criterion(predicted, target)
print("MSE Loss:", loss.item())
```

```
Predicted:  tensor([[-0.9763,  0.7022],
        [ 1.2322, -0.0076]])
Target:  tensor([[ 0.1120, -2.2266],
        [-0.6141, -0.1838]])
MSE Loss: 3.3005387783050537
```

Cross-Entropy Loss (nn.CrossEntropyLoss)

Cross-Entropy loss is a common choice for classification tasks. It calculates the log likelihood of the correct label, which can be seen as a measure of 'uncertainty'.

- **Pros:** It works well for multi-class classification tasks. It penalizes confident and wrong predictions heavily.
- **Cons:** It might produce a large gradient update when the model is very confident but wrong, causing unstable learning dynamics.
- **Common Usage:** Classification tasks.

```python
import torch
import torch.nn as nn

predicted = torch.randn(2, 2)
target = torch.randn(2, 2)
print("Predicted: ", predicted)
print("Target: ", target)

criterion = nn.CrossEntropyLoss()

loss = criterion(predicted, target)
print("MSE Loss:", loss.item())
```

```
Predicted:  tensor([[-1.6950,  1.1574],
        [-0.4759, -0.9480]])
Target:  tensor([[-0.8139,  1.9506],
        [ 0.2033, -0.2159]])
MSE Loss: -1.1828951835632324
```

Binary Cross-Entropy Loss (BCELoss)

Binary Cross-Entropy loss is a special case of Cross-Entropy loss that is used for binary classification tasks.

- **Pros:** It's suitable for binary classification problems and works well when classes are imbalanced.
- **Cons:** Similar to CrossEntropyLoss, it can also produce a large gradient update when the model is confident and wrong.
- **Common Usage:** Binary classification tasks.

```python
import torch
import torch.nn as nn

predicted = torch.randn(2, 2)
target = torch.randn(2, 2)
print("Predicted: ", predicted)
print("Target: ", target)

sigmoid = nn.Sigmoid()
predicted = sigmoid(predicted)
criterion = nn.BCELoss()

loss = criterion(predicted, target)
print("BCE Loss:", loss.item())
```

```
Predicted:  tensor([[-0.4528, -1.1972],
        [-0.3271,  1.4739]])
Target:  tensor([[1.9719, 0.7932],
        [0.3363, 0.0883]])
BCE Loss: 1.2003655433654785
```

Binary Cross-Entropy with Logits Loss (BCEWithLogitsLoss)

This loss function combines a Sigmoid layer and the BCELoss in one single class. It is more numerically stable than using a plain Sigmoid followed by a BCELoss.

- **Pros:** It's more numerically stable than using a plain Sigmoid followed by a BCELoss as it handles the logits directly.
- **Cons:** It's specifically for binary classification, not suitable for multi-class classification or regression.
- **Common Usage:** Binary classification tasks with real-valued predictions (logits).

```python
import torch
import torch.nn as nn

predicted = torch.randn(2, 2)
target = torch.randn(2, 2)
print("Predicted: ", predicted)
print("Target: ", target)

criterion = nn.BCEWithLogitsLoss()

loss = criterion(predicted, target)
print("BCE Loss:", loss.item())
```

```
Predicted:  tensor([[-1.1919,  0.0773],
        [ 0.8714,  2.3575]])
Target:  tensor([[ 0.0036, -0.0941],
        [-0.5097, -0.2015]])
BCE Loss: 1.3993690013885498
```

L1 Loss (L1Loss)

L1 Loss is used for regression tasks and it computes the absolute differences between the target and the predicted values. It is more robust to outliers than MSE.

- **Pros:** It's more robust to outliers than MSE. It provides sparse solutions, i.e., promotes zero residuals which can serve as a feature selector.
- **Cons:** It's not differentiable at zero, and hence, optimization could be harder than with the MSE.
- **Common Usage:** Regression tasks, especially when you want to resist outliers in the data.

```python
import torch
import torch.nn as nn

predicted = torch.randn(2, 2)
target = torch.randn(2, 2)
print("Predicted: ", predicted)
print("Target: ", target)

criterion = nn.L1Loss()

loss = criterion(predicted, target)
print("BCE Loss:", loss.item())
```

```
Predicted:  tensor([[0.0313, 0.7780],
        [0.6076, 0.2979]])
Target:  tensor([[-0.6857,  0.2483],
        [-0.0690, -0.1353]])
BCE Loss: 0.5891140699386597
```