

Advanced NLP: Assignment 3

Hitesh Goel 

2020115003 

OneDrive link for pth files: 

[Link](#)


It has the following contents:

- `models/` : All the `pth` files i.e. all the 11 transformers trained. They have been named according to the hyperparameters used.
- `out_files` : Sentence wise bleu scores for the train and test set for each model.
- `log_files` : Log files for all the 11 transformer models.

Training the transformer:

```
cd src
```

```
python main.py --lr <learning_rate> --bs <batch_size> --eps <number_of_epochs> --dr
```

- 
- batch size: Only ≤ 8 size works for 1 GPU on Ada. Default is also set to 8.
 - number of epochs: 1 epoch takes about 20-30 minutes when you submit a batch job on Ada. Default is set to 10.
 - dropout: set accordingly, default = 0.1
 - attention heads: model dimension is 512, so set accordingly. default number is 8. (same as the attention is all you need paper)
 - layers: default number is 6 as per the attention is all you need paper. feel free to inc/dec.

Theory

Question 1:

What is the purpose of self-attention, and how does it facilitate capturing dependencies in sequences?

Answer 1:

In their paper, *Attention Is All You Need*, Vaswani et al. (2017) explain that the Transformer model, relies solely on the use of self-attention, where the representation of a sequence (or sentence) is computed by relating different words in the same sequence.

Attention: Attention is a mechanism that allows a model to focus on specific parts of the input (or sequence) when processing the data. It assigns weights to different elements, emphasizing or de-emphasizing them based on their relevance to the current processing step. This weighted focus is especially crucial in cases where not all parts of the input are equally relevant for the task at hand.

The main components of attention are:

- q and k denoting vectors of dimension d_k containing the queries and keys, respectively.
- v denoting a vector of dimension d_v containing the values.
- Q , K and V denoting matrices packing together sets of queries, keys, and values, respectively.
- w_q , w_k , w_v denoting projection matrices that are used in generating different subspace representations of the query, key, and value matrices.
- w_o denoting a projection matrix for the multi-head output.

In essence, the attention function can be considered a mapping between a query and a set of key-value pairs to an output.

Attention is calculated as:

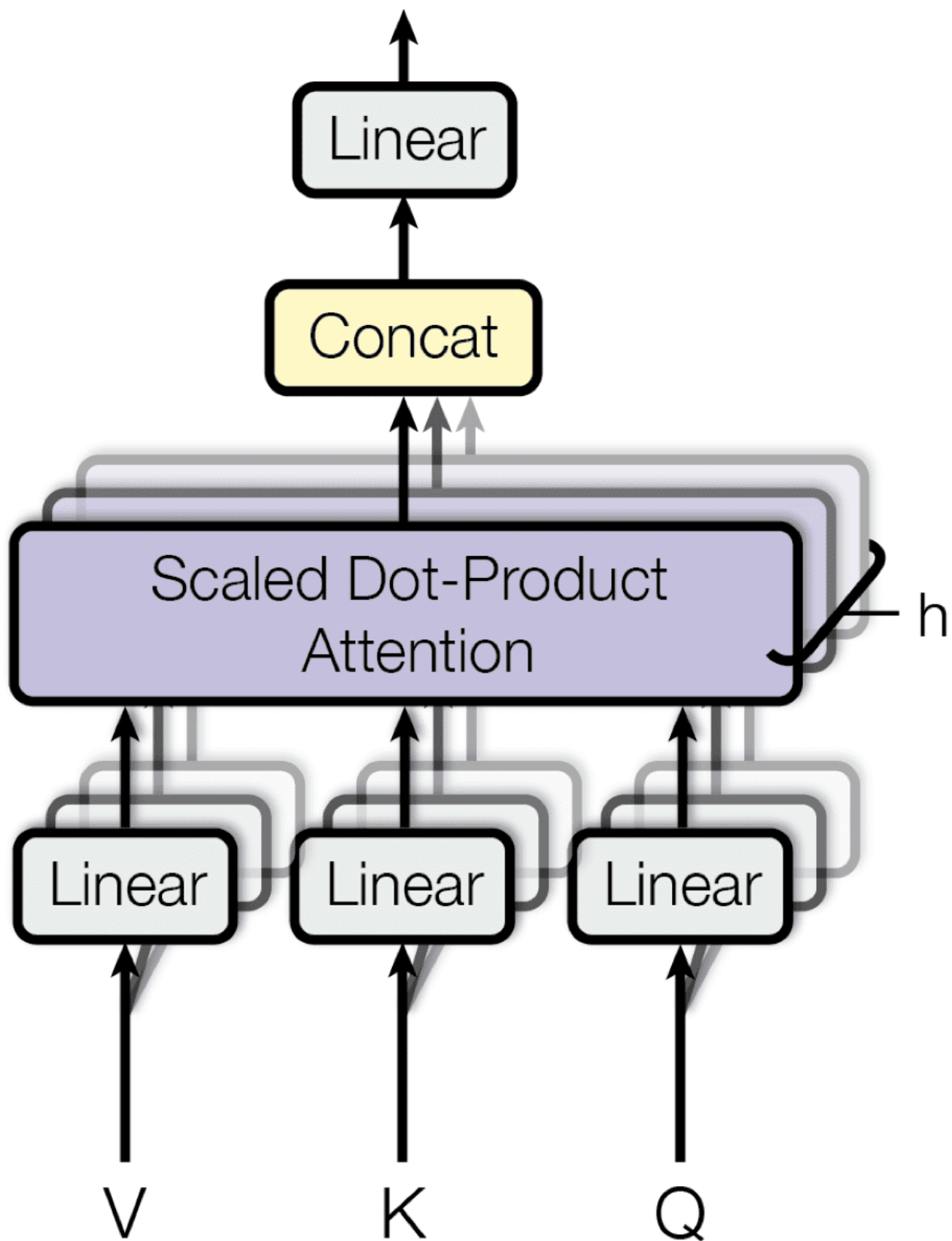
$$\text{attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V$$

This scaling factor was introduced to counteract the effect of having the dot products grow large in magnitude for large values of d_k , where the application of the softmax function would then return extremely small gradients that would lead to the infamous vanishing gradients problem. The scaling factor, therefore, serves to pull the results generated by the dot product multiplication down, preventing this problem.

Self Attention: Self-attention, also known as intra-attention, is a special case of attention where the input sequence is used as both the source of queries, keys, and values. In other words, q , k , and v all come from the same input sequence. Self-attention allows each element in the sequence to focus on every other element, capturing dependencies and relationships within the sequence.

Multi-Head Attention: In multi-head attention, the self-attention mechanism is applied in parallel with multiple sets of learned parameters, known as "heads". Each head learns different sets of q , k , and v parameters, producing different representations. These head outputs are then concatenated and linearly projected to obtain the final multi-head attention output. This enables the model to learn multiple aspects or patterns in the data simultaneously.

In the paper, *Attention is All You Need*, their multi-head attention mechanism linearly projects the queries, keys, and values n times, using a different learned projection each time. The single attention mechanism is then applied to each of these n projections in parallel to produce n outputs, which, in turn, are concatenated and projected again to produce a final result.



Facilitating Dependency Capture: By considering all words in the input sequence and calculating attention scores for each word, self-attention allows the model to capture both short and long-range dependencies. The model can give higher attention to words that are crucial for understanding the context or relationships within the sequence, regardless of their position. Self-attention can be enhanced by using multiple attention heads and stacking multiple layers of attention mechanisms. Each head provides a different perspective on how to weigh the words, enabling the model to capture a broader range of dependencies and patterns.

In summary, self-attention enables the model to weigh the relevance of each word in the input sequence concerning every other word. This allows the model to capture complex and non-local dependencies, making it highly effective for tasks involving understanding and generating sequences, such as machine translation, summarization, question-answering, and more.

Question 2: 🔗

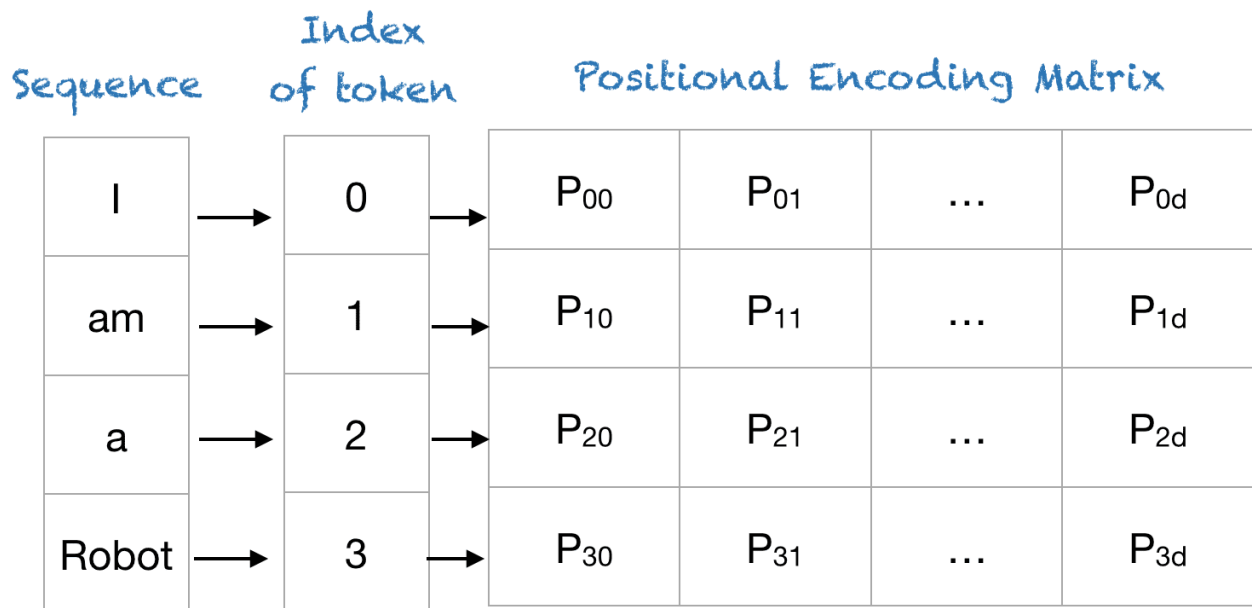
**Why do transformers use positional encodings in addition to word embeddings?
Explain how positional encodings are incorporated into the transformer architecture.**

Answer 2: 🔗

In languages, the order of the words and their position in a sentence really matters. The meaning of the entire sentence can change if the words are re-ordered. When implementing NLP solutions, recurrent neural networks have an inbuilt mechanism that deals with the order of sequences. The transformer model, however, does not use recurrence or convolution and treats each data point as independent of the other. Hence, positional information is added to the model explicitly to retain the information regarding the order of words in a sentence. Positional encoding is the scheme through which the knowledge of the order of objects in a sequence is maintained.

What is Positional Encoding? Positional encoding describes the location or position of an entity in a sequence so that each position is assigned a unique representation. There are many reasons why a single number, such as the index value, is not used to represent an item's position in transformer models. For long sequences, the indices can grow large in magnitude. If you normalize the index value to lie between 0 and 1, it can create problems for variable length sequences as they would be normalized differently.

Transformers use a smart positional encoding scheme, where each position/index is mapped to a vector. Hence, the output of the positional encoding layer is a matrix, where each row of the matrix represents an encoded object of the sequence summed with its positional information. An example of the matrix that encodes only the positional information is shown in the figure below.



Positional Encoding Matrix for the sequence 'I am a robot'

Positional Encoding Layer in Transformers: Let's dive straight into this. Suppose you have an input sequence of length L and require the position of the k th object within this sequence. The positional encoding is given by sine and cosine functions of varying frequencies:

$$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P(k, 2i + 1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

Here,

- k : Position of an object in the input sequence, $0 \leq k \leq L/2$
- d : Dimension of the output embedding space
- $P(k, j)$: Position function for mapping a position k in the input sequence to index (k, j) of the positional matrix.
- n : User-defined scalar, set to 10,000 by the authors of *Attention Is All You Need*.

- i : Used for mapping to column indices $0 \leq i < d/2$, with a single value of i maps to both sine and cosine functions.

In the above expression, you can see that even positions correspond to a sine function and odd positions correspond to cosine functions.

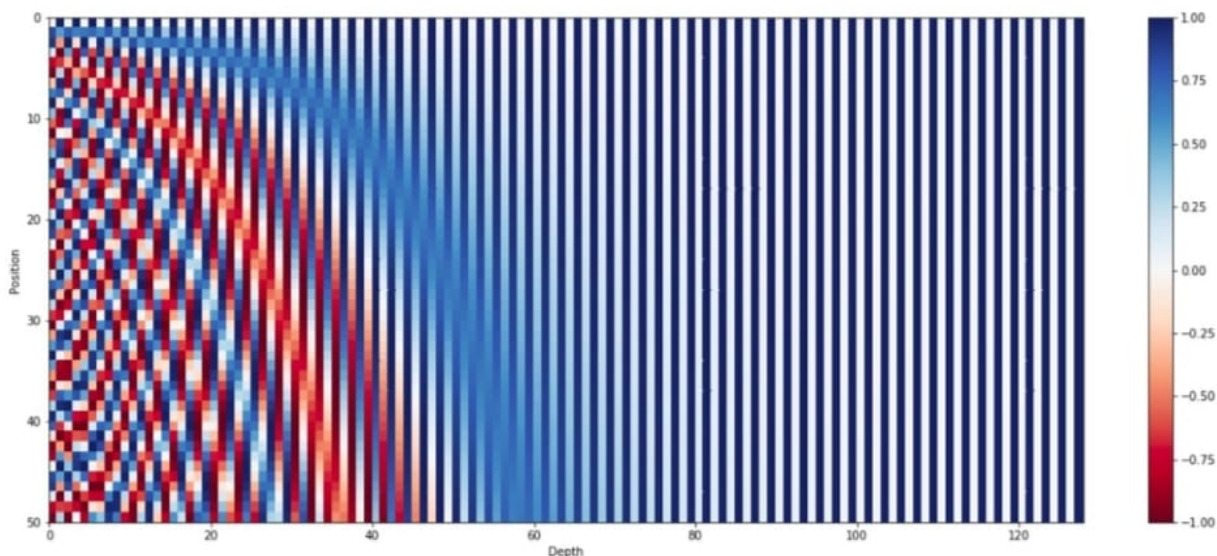


Figure 2 - The 128-dimensional positional encoding for a sentence with the maximum length of 50. Each row represents the embedding vector \vec{p}_t

Sequence	Index of token, k	Positional Encoding Matrix with $d=4$, $n=100$			
		$i=0$	$i=0$	$i=1$	$i=1$
I	0	$P_{00}=\sin(0)$ = 0	$P_{01}=\cos(0)$ = 1	$P_{02}=\sin(0)$ = 0	$P_{03}=\cos(0)$ = 1
am	1	$P_{10}=\sin(1/1)$ = 0.84	$P_{11}=\cos(1/1)$ = 0.54	$P_{12}=\sin(1/10)$ = 0.10	$P_{13}=\cos(1/10)$ = 1.0
a	2	$P_{20}=\sin(2/1)$ = 0.91	$P_{21}=\cos(2/1)$ = -0.42	$P_{22}=\sin(2/10)$ = 0.20	$P_{23}=\cos(2/10)$ = 0.98
Robot	3	$P_{30}=\sin(3/1)$ = 0.14	$P_{31}=\cos(3/1)$ = -0.99	$P_{32}=\sin(3/10)$ = 0.30	$P_{33}=\cos(3/10)$ = 0.96

Positional Encoding Matrix for the sequence 'I am a robot'

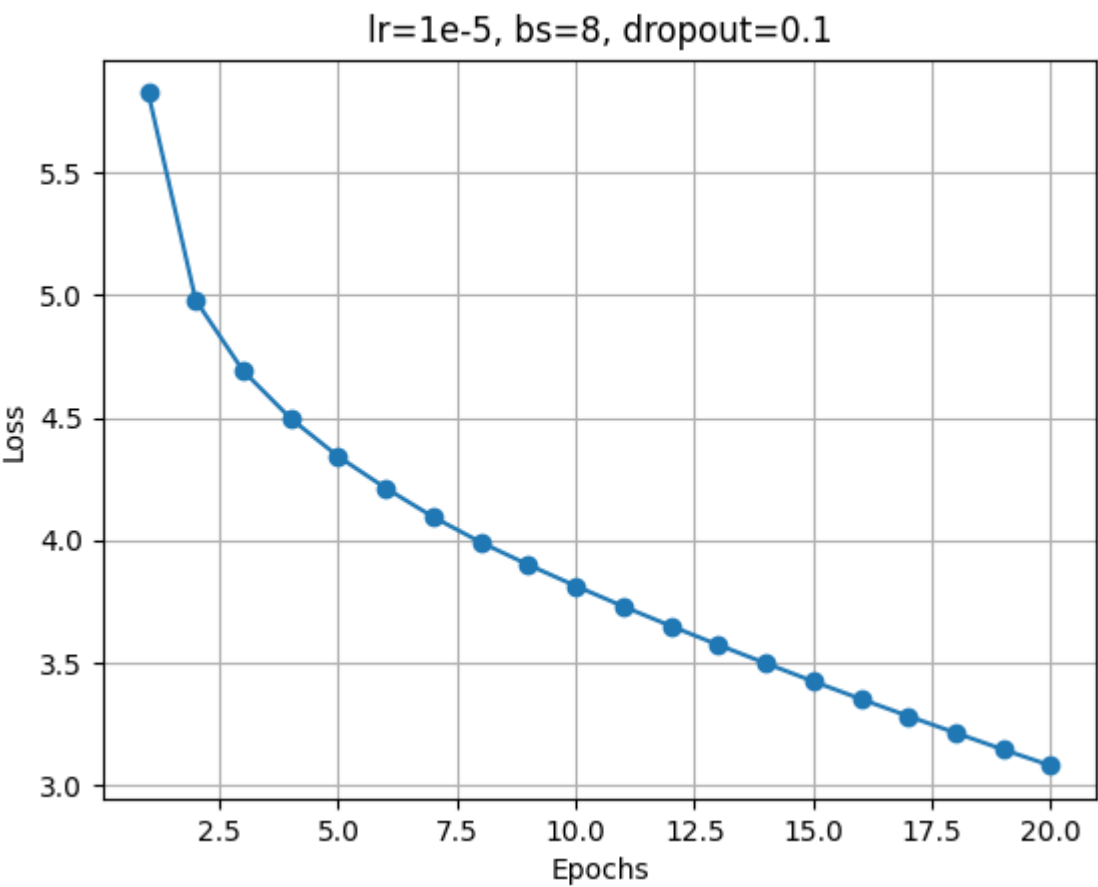
Hyperparameter Finetuning and Analysis

1. Varying the Learning Rate

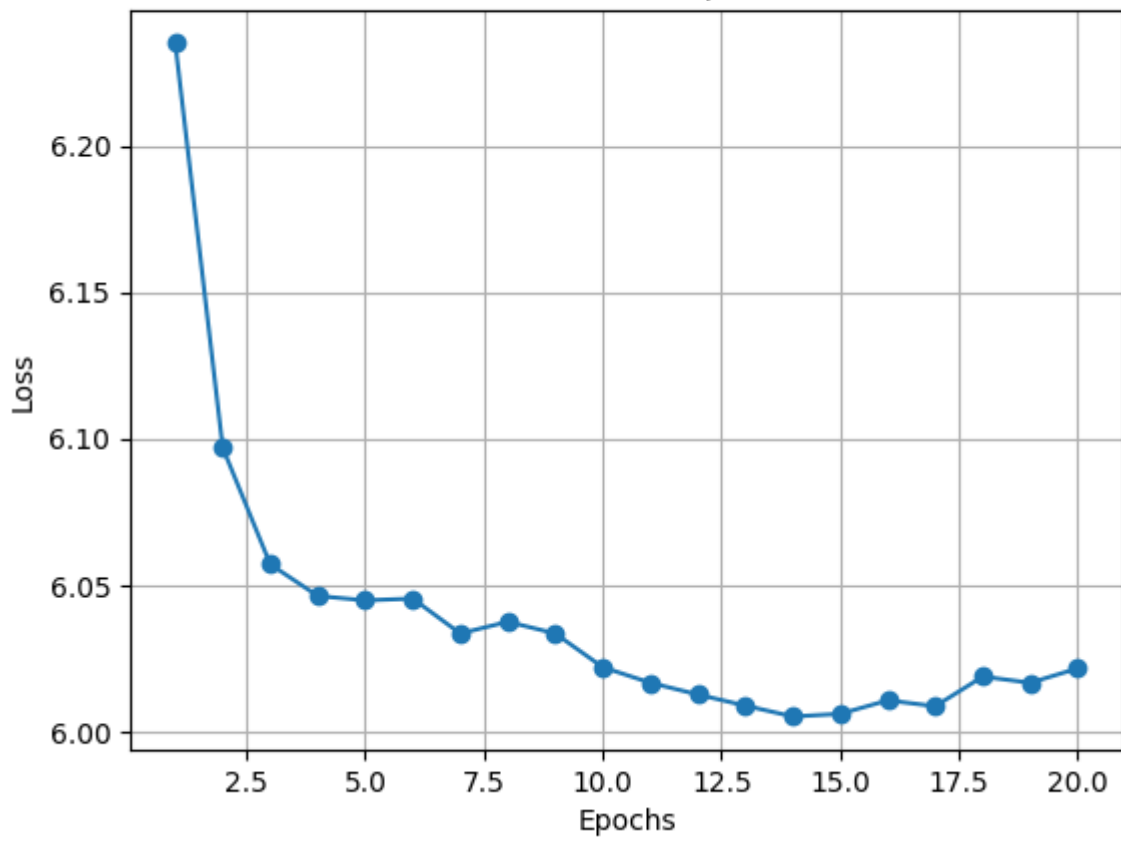
Results Obtained:

Model	Learning Rate	Batch Size	Dropout	Layers	Best Loss	Bleu Score (Train)	Bleu Score (Test)
1	1e-5	8	0.10	6	3.0820	0.0805	0.0796
2	5e-4	8	0.10	6	6.0218	0	0
3	1e-4	8	0.10	6	0.6528	0.0893	0.0877
4	1e-3	8	0.10	6	5.9934	0	0

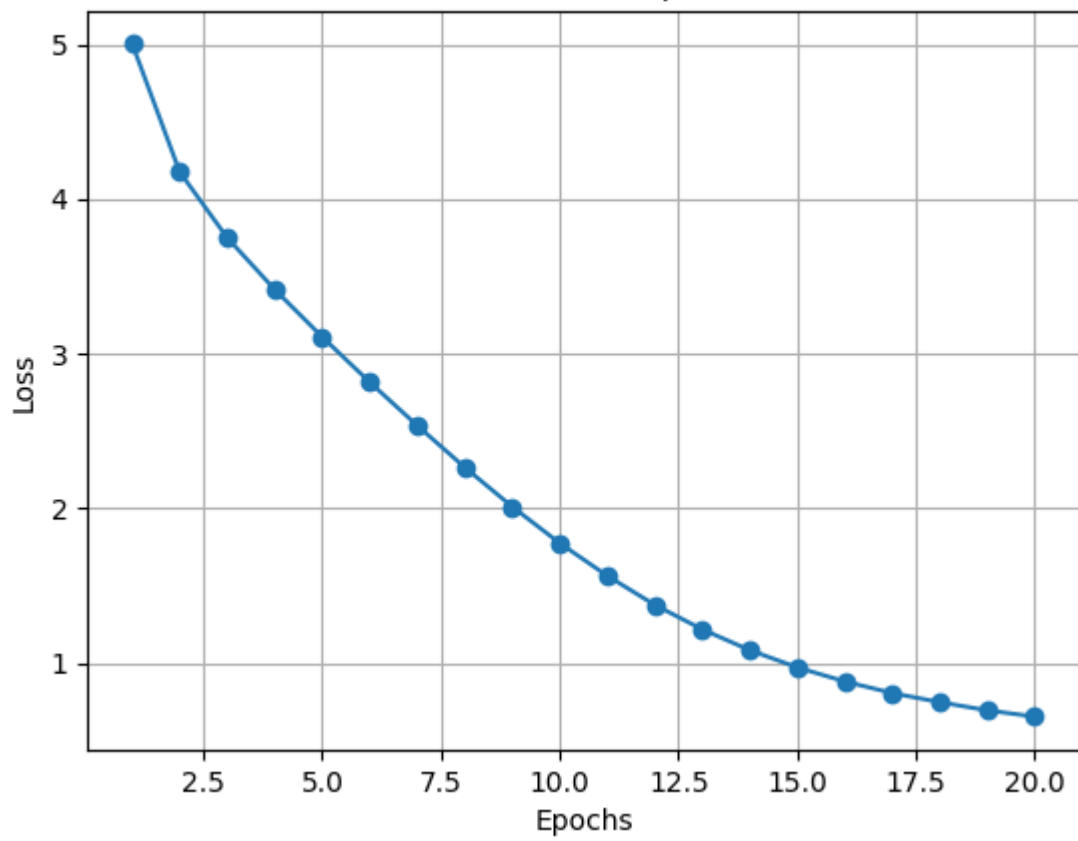
Loss Plots for each model:

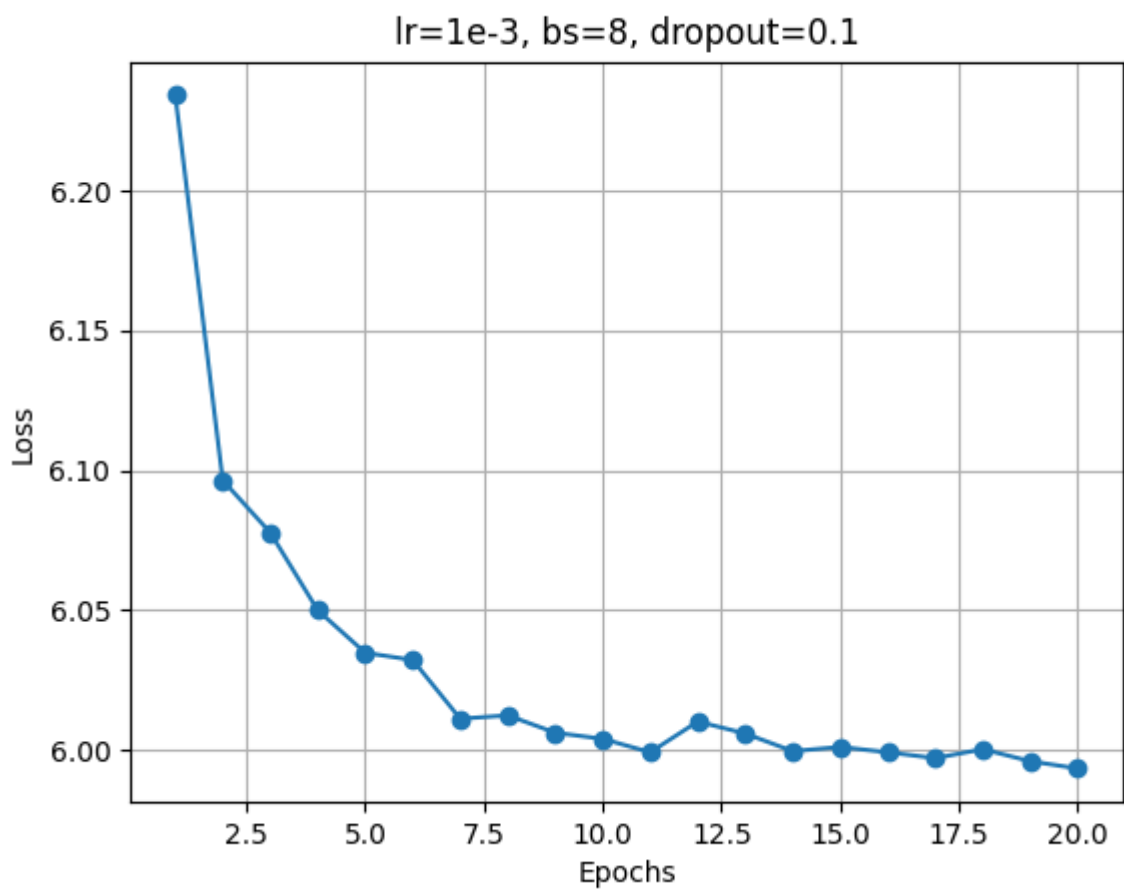


lr=5e-4, bs=8, dropout=0.1

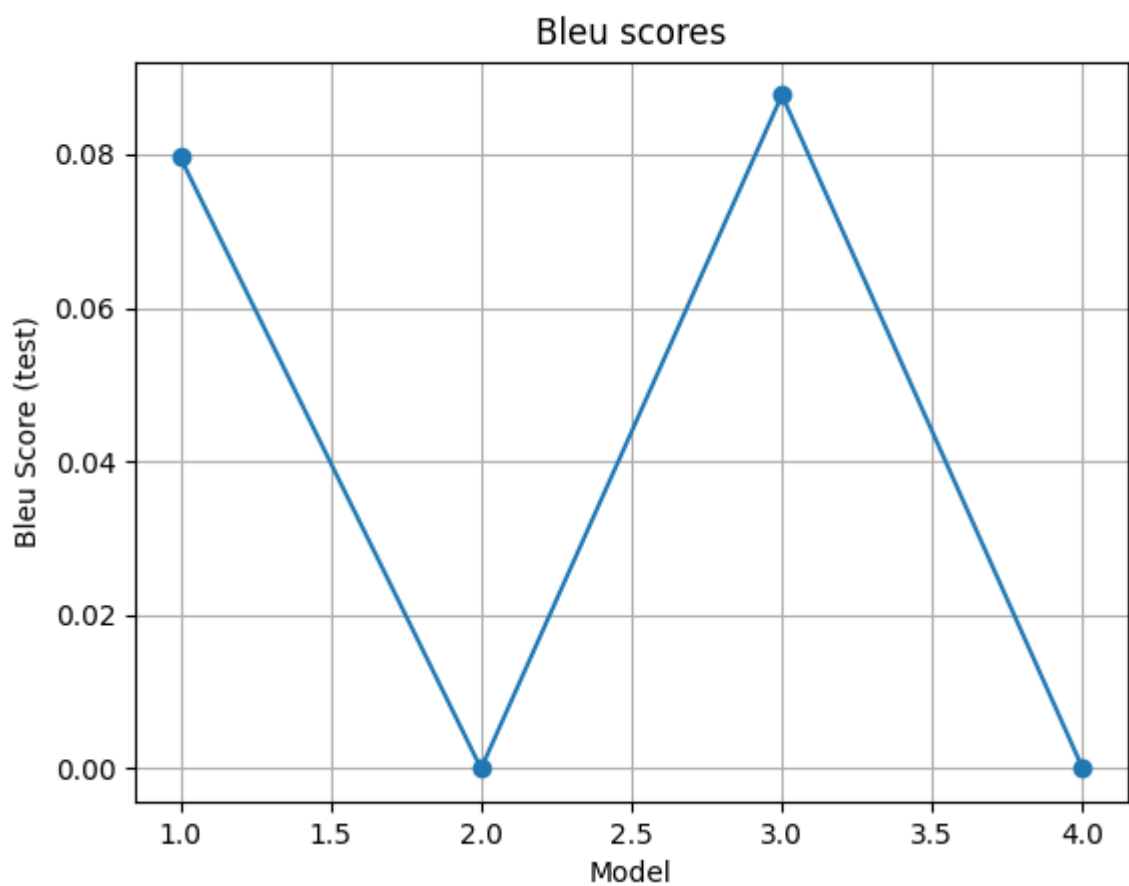


lr=1e-4, bs=8, dropout=0.1





Bleu score plot: [🔗](#)



What do we learn? 🔗

- Model 1 and Model 3 with $lr = [1e-5, 1e-4]$ show continuous decrease in loss.
- Meanwhile the other 2 models with $lr = [5e-4, 1e-3]$ have loss curves that increase/decrease after each epoch randomly.
- This shows that $lr > 1e-4$ is *harmful for training a large model* like the transformer.
- This could also be because of the small size of the training data. A high learning rate means that the loss does not converge well and we never reach the optimal minima.
- $lr = 1e-4$ gives the best results out of all the models.
- $lr = 1e-5$ performs slightly worse because the lr is probably too low and hence the function takes a *longer time to converge to the minima*, on the other hand, $1e-4$ achieves the minima quicker.

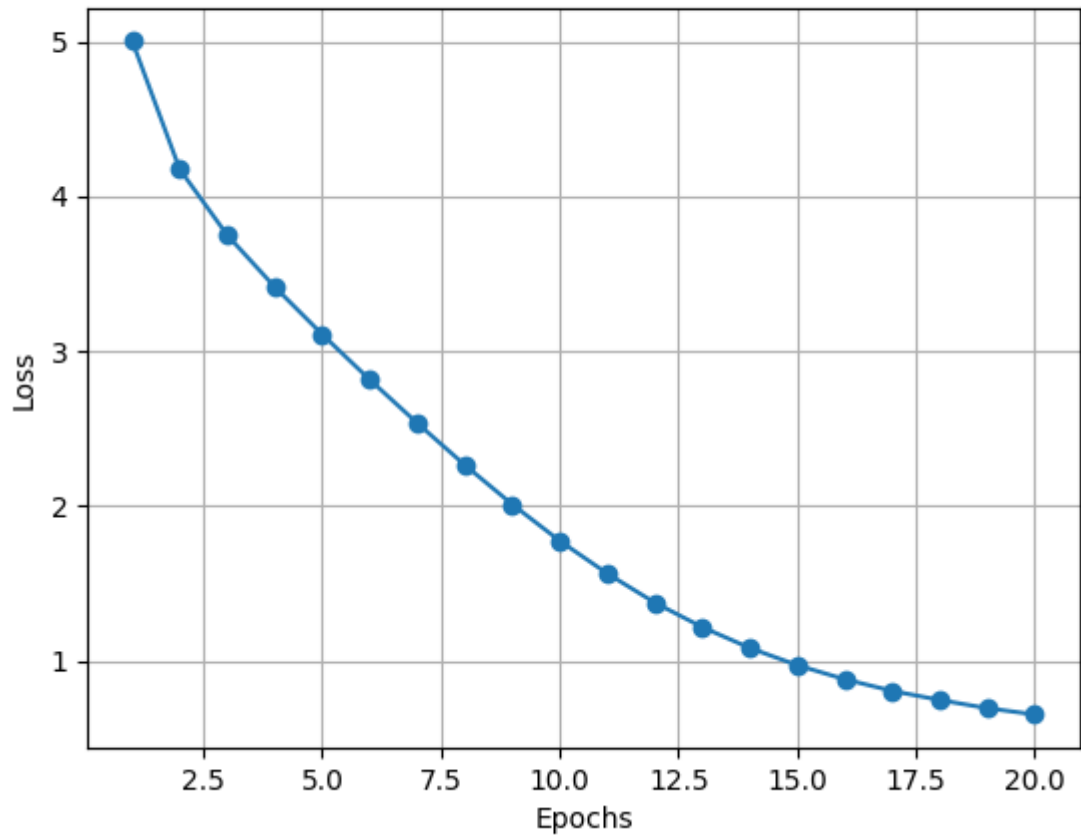
2. Varying dropout 🔗

Results Obtained: 🔗

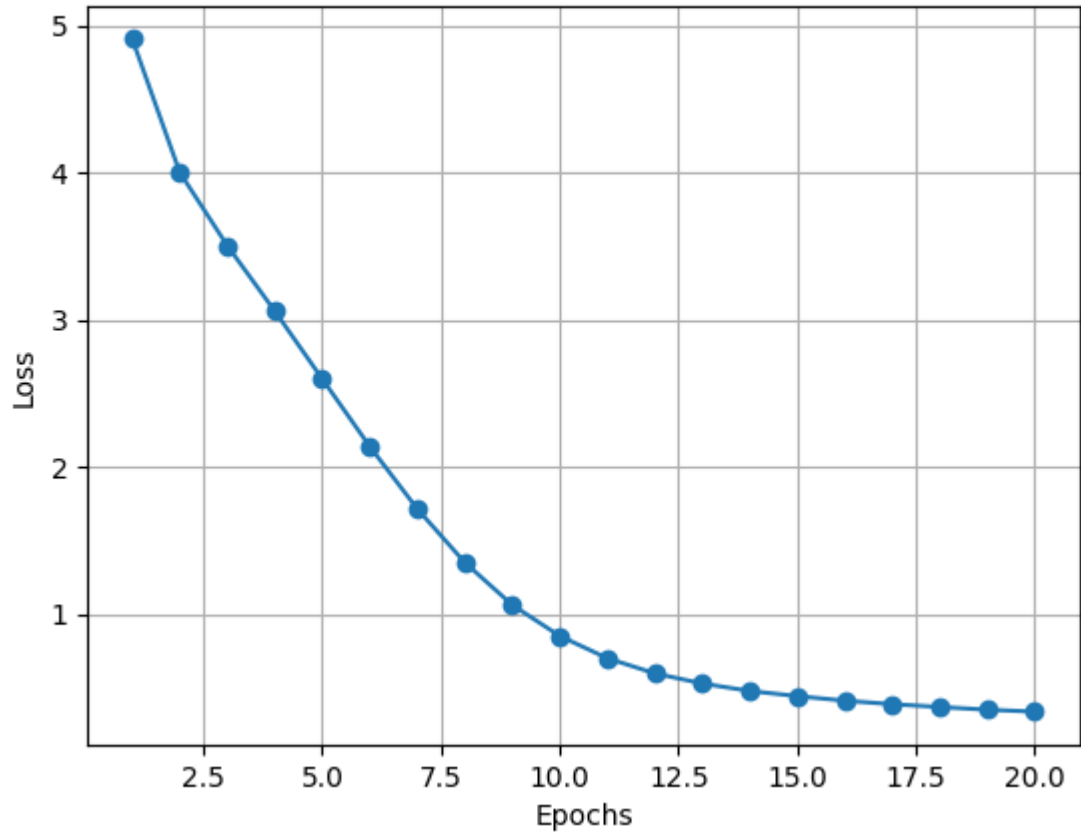
Model	Learning Rate	Batch Size	Dropout	Layers	Best Loss	Bleu Score (Train)	Bleu Score (Test)
1	1e-4	8	0.10	6	0.6528	0.0893	0.0877
2	1e-4	8	0.00	6	0.3350	0.0883	0.0778
3	1e-4	8	0.15	6	0.7927	0.0681	0.0675
4	1e-4	8	0.20	6	1.0505	0.0680	0.0672

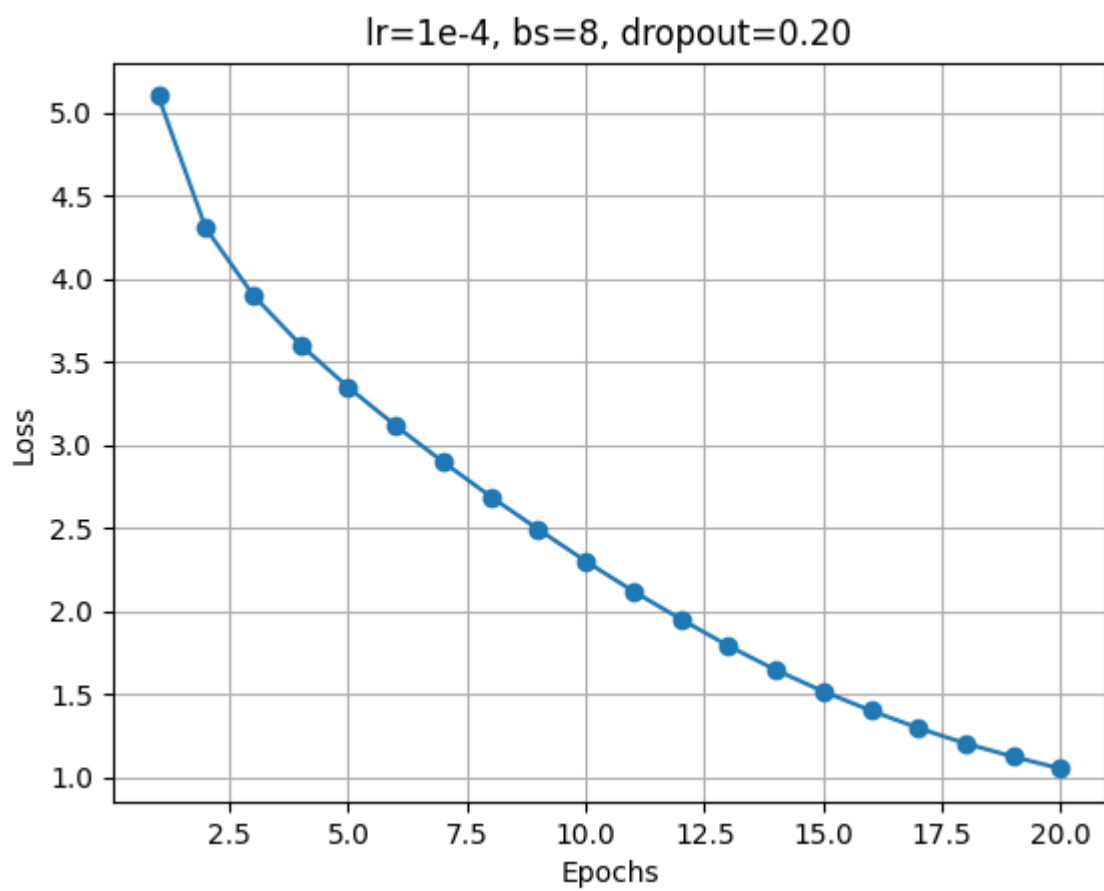
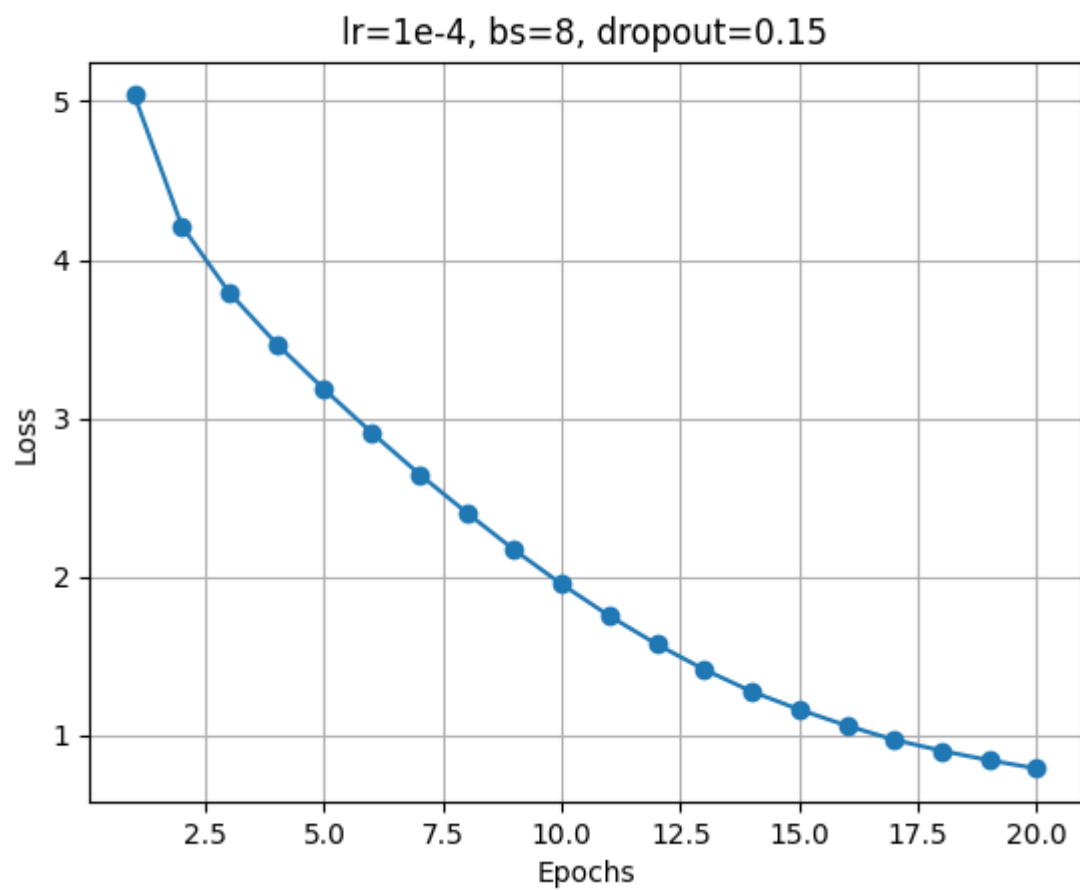
Loss plots for each model 🔗

lr=1e-4, bs=8, dropout=0.1

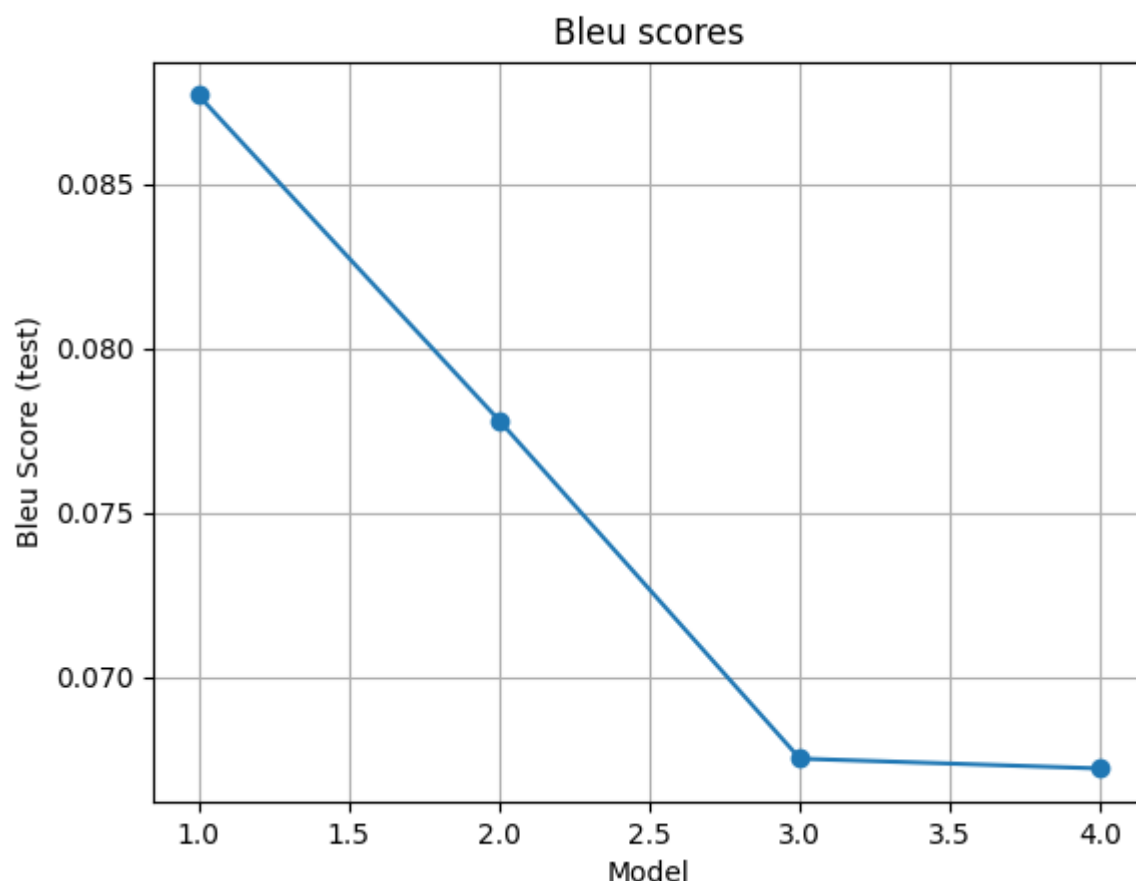


lr=1e-4, bs=8, dropout=0.0





Bleu score plot: [↗](#)



What do we learn?

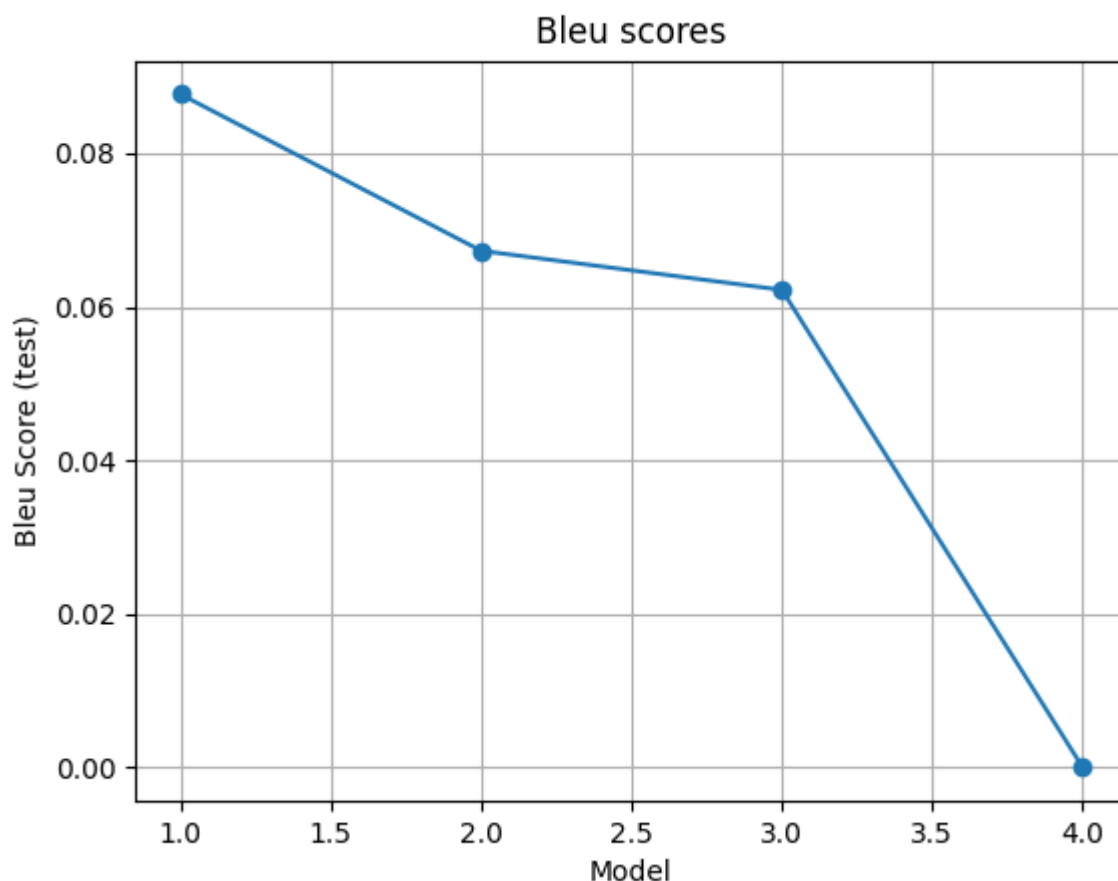
- *dropout=0.10 gives the best performance* followed by dropout=0.
- Dropout is important because it *freezes some of the layers randomly while training and this gives us an ensemble of models* that helps the model train and predict better.
- However, as we can see freezing too many layers is also harmful to the health of the model, possibly because it is not able to learn correctly.

3. Varying batch size

Results Obtained:

Model	Learning Rate	Batch Size	Dropout	Layers	Best Loss	Bleu Score (Train)	Bleu Score (Test)
1	1e-4	8	0.10	6	0.6528	0.0893	0.0877
2	1e-4	4	0.10	6	1.3721	0.0680	0.0673
3	1e-4	2	0.10	6	1.6891	0.0629	0.0622
4	1e-4	1	0.10	6	Nan	0	0

Bleu score plot:



What do we learn?

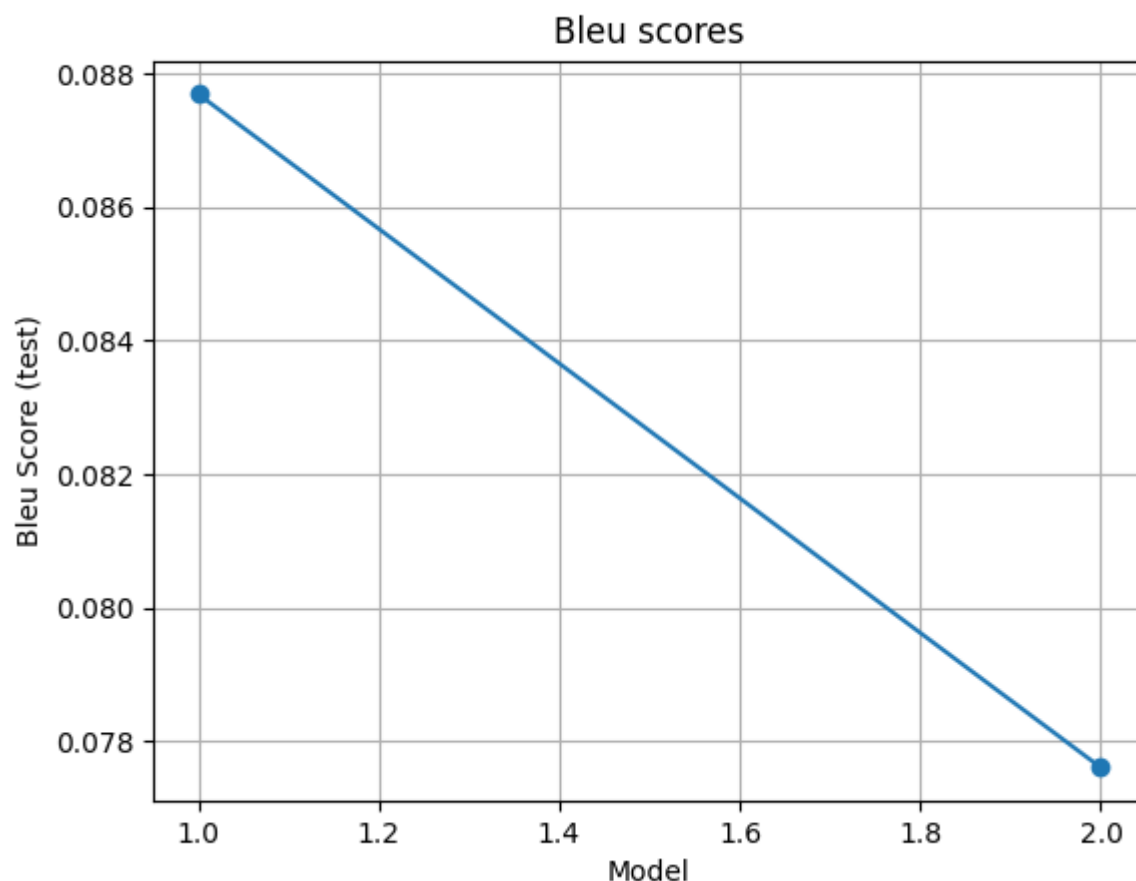
- Batch size ≥ 8 does not fit on Ada for 1 GPU.
- The *performance of the model seems to increase as the batch size increases*.
- Batch size=8 gives us the best performance since that is the max size we can fit on Ada.
- Batch size=1 does not train at all.
- Batch size = [4,2] also give decent performances, but they are much worse than bs=8 and also take a *longer time to train*.

4. Varying number of layers

Results obtained:

Model	Learning Rate	Batch Size	Dropout	Layers	Best Loss	Bleu Score (Train)	Bleu Score (Test)
1	1e-4	8	0.10	6	0.6528	0.0893	0.0877
2	1e-4	8	0.10	2	1.0723	0.0781	0.0776

Bleu score plot:



What do we learn?

- Since the training set was small, the idea was to reduce the size of the model to see if it gives better performance.
- While the model with 2 layers takes *much less time to train*, the model with 6 *encoder and decoder layers still performs better*.
- Thus the standard architecture mentioned in the *Attention is all you need* paper outperforms the other model.

Overall best model obtained

The best model has the following parameters:

- $lr = 1e-4$ $bs = 8$ dropout = 0.10 layers = 6

The model obtained the following scores:

1. Best loss achieved: 0.6528
2. Bleu score for the train corpus: 0.0893
3. Bleu score for the test corpus: 0.0887

- The analysis of the hyperparameters chosen has already been done in the previous sections. Loss curves have also been attached.
- The bleu scores for the sentences are available in the *OneDrive Link* given at the start of the document.

References

- [link 1](#)
- [link 2](#)
- [link 3](#)
- [link 4](#)