OPTIMIZING MACHINE TRANSLATION MODEL: A HYPERPARAMETER ABLATION STUDY

Team 9: Ketaki Kolhatkar[a], Luv Verma[b]

[a]*Khoury College of Computer Sciences, Northeastern University, Boston,*
[b]*Khoury College of Computer Sciences, Northeastern University, Boston,*

Abstract

In machine translation tasks, the relationship between model complexity and performance is often presumed to be linear, driving an increase in the number of parameters and consequent demands for computational resources like multiple GPUs. This study challenges this assumption by systematically exploring the effects of hyperparameters through ablation on a sequence-to-sequence machine translation pipeline, utilizing a single NVIDIA A100 GPU. Contrary to expectations, our experiments reveal that combinations with the most parameters were not necessarily the most effective. This unexpected insight prompted a careful reduction in parameter sizes, uncovering "sweet spots" that enable training sophisticated models on a single GPU without compromising translation quality. The findings demonstrate an intricate relationship between hyperparameter selection, model size, and computational resource needs. The insights from this study contribute to the ongoing efforts to make machine translation more accessible and cost-effective, emphasizing the importance of precise hyperparameter tuning over mere scaling. To start with the project of multiprocessing a model on GPU, we began by understanding the basics of multiprocessing. There are multiple ways of working with CUDA: CUDA C/C++: most common and flexible way to use CUDA while writing code in C/C++ pyCUDA: gives us complete access to CUDA C/C++ API in Python code. We would still have to write the C/C++ code in Python. Numba: this method does not expose the entire CUDA C/C++ API, yet allows acceleration with very less modifications while writing the code in Python itself

Keywords: Natural Language Processing (NLP), Python Acceleration, CUDA, Transformers, Machine Translation, Ablation

1. Introduction

The advent of technology has propelled humankind into a new era where machine learning is progressing rapidly. With every passing day, models in fields such as data engineering, finance, computer vision, and natural language processing are growing both in size and capability. However, a pivotal question arises: Is bigger always better? In our current studies, we approached this question from two perspectives. First, we delved into the intricacies of multiprocessing within Python using CUDA [1]. Second, we selected a complex machine translation task in the domain of natural language processing, requiring a complete sequence-to-sequence pipeline [2], to examine the relationship between model size and performance. Traditionally, the enhancement of model performance has been synonymous with an increase in parameters, layers, and complexity. Yet, we must confront the reality that the relationship between size and efficacy is not necessarily linear, especially when considering the need for efficiency, stability, and resource-conscious design. Our journey began with an ambitious plan to utilize both model parallelism and data parallelism in PyTorch, with a starting model comprising 500 million parameters. However, constraints such as the availability of multi-GPUs and a limited compute window of 8 hours on a single A100 GPU [3] steered us toward a different path. Instead of continually expanding the model, we embarked on a gritty exploration to reduce the model's parameters. We conducted ablation studies, manipulating variables like model size, number of heads, layers, and dropouts, with a computation limit on an A100 single GPU. By capping the number of epochs to 100, we investigated the effects of varying parameter sizes on the accuracies of machine translation models. Our findings culminated in the fascinating observation that an increase in parameters does not always yield a better model, a notion aligned with recent research conducted in Chinchilla paper [4]. This study underscores the idea that training a robust model does not necessarily require a multi-GPU setup. Before scaling up hardware, it is vital to meticulously analyze how the model is performing within existing resources and understand how the model's size correlates with accuracy.

2. Python Accelerated Experiments

We have explored the Numba method in this project in detail. Analyzing this method helped us learn multiprocessing in Python without using C/C++ code. Numba is a just-in-time function compiler for accelerating numeric Python functions for GPU and CPU. To get started exploring Numba we write a hypotenuse function with the jit function decorator - used for Numba's CPU compilation. Using the *%timeit* helped us measure the performance of our code while using the Numba component. The $pyfunc$ function allowed us to measure the per-

Function	CPU/Python	Time Taken
Hypotenuse	Python version	693 ns ± 7.88 ns/loop
Hypotenuse	CPU	190 ns ± 0.0269 ns/loop
Hypotenuse	in-built Python	139 ns ± 0.031 ns/loop

Table 1: Hypotenuse function on CPU accelerated/Python

Function	Numpy/Numpy	Time Taken
Addition	Numpy on CPU	1.03 µs ± 0.24 ns/loop
Addition	Numba on GPU	688 µs ± 1.05 µs/loop

Table 2: Addition function on Numpy/Numba

formance of the pure Python version of code without the CPU accelerated part. The experiments performed on the Python version accelerated CPU version and the in-built Python version and their time taken have been shown in Table 1.

Numba's type names mirror Numpy's type names like Python float is called a float64. Although Numba supports many functions, it does not support all Python functions. One of the functions Numba can not compile is dictionaries. Numba's jit goes into an object or a default mode when it does not support a function. Similar to Numpy's vectorize, Numba has the vectorize decorator for scalar inputs and broadcasts it. Vectorize decorator requires explicit type signatures and target defined. This comparison of the time taken on Numpy and Numba has been shown in Table 2.

Numba automatically does the following things when calling this function with vectorize:
1. Compiled a CUDA kernel to execute the ufunc operation in parallel over all input elements
2. Allocated GPU memory for the inputs and the output
3. Copied the input data to the GPU
4. Executed the CUDA kernel (GPU function) with correct kernel dimensions given input sizes
5. Copied the result back from the GPU to the CPU
6. Returned the result as a NumPy array on the host

Surprisingly the GPU version code works much slower than the CPU version. This slow GPU performance of the code might happen due to the misuse of the GPU. To explore the reasons why we might have misused the GPU are: GPU uses parallelism on large inputs, processes thousands of values at once and achieves a good performance. But given our small input size, it is not enough to keep the GPU busy. If our calculations are too simple and do not have enough math operators, then the GPU spends most time waiting for the data to move around. It involves quite a bit of overhead compared to calling a function on the CPU. In most cases paying the cost of moving the data to and from the GPU might be worth processing the function. While for simpler functions moving the data around might not be worth it due to its simple functionality. Our example uses int64 when we probably don't need it. Scalar code using data types that are 32 and 64-bit run basically the same speed on the CPU, and for integer types the difference may not be drastic, but 64-bit floating point data types may have a significant performance cost on the GPU, depending on the GPU

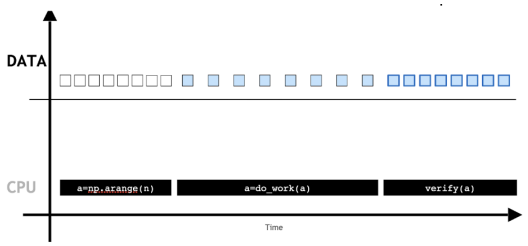

Figure 1: CPU-Accelerated: Data is allocated on the CPU and work is performed serially on the CPU

Function	CPU/Python	Time Taken
Scalar Addition	Host device	693 ns ± 7.88 ns/loop
Scalar Addition	CUDA functions	567 µs ± 354 ns/loop
Scalar Addition	CUDA out device	455 µs ± 623 ns/loop

Table 3: Hypotenuse function on CPU accelerated/Python

type. The comparison of scalar addition on CPU, CUDA and CUDA with an out device along with the time taken for each of them has been shown in Table 3.

2.1. Grid Stipe Loop

Inside kernel definitions, CUDA-provided variables describe its executing thread, block and grid as shown in Figure 3:
gridDim.x: the number of blocks in the grid (here we have 2)
blockIdx.x: the index of the current block within the grid (0, 1)
blockDim.x: the number of threads in a block (here 4)
threadIdx.x: the index of the thread within a block (0, 1, 2, 3)
Assume data is in a 0-indexed vector, somehow each thread must be mapped to work on elements in the data. If we calculate a thread's index within the entire grid, then we could map that index to an index in the data. CUDA does not provide a single variable to capture this, thread indices within the block there is an idiomatic way to calculate the value. Each thread has access to the size of its block via blockDim.x and the index within the grid via blockIdx.x and its own index within its block via threadIdx.x. Using these variables, the formula threadIdx.x + blockIdx.x * blockDim.x will return the thread's unique index in the whole grid, which we can map to data ele-

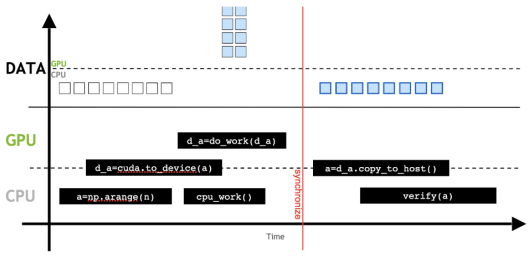

Figure 2: GPU-Accelerated: In accelerated applications both host and device memory are used, data can be initialized on the CPU and copied to the GPU device where it can be worked on in parallel. GPU work is asynchronous to the host so both CPU and GPU work can happen simultaneously. Synchronization points between CPU and GPU can be indicated using cuda.synchronize() function - data is copied back to the CPU.

Function	CPU/GPU	Time Taken
Monte Carlo Pi	CPU	108 ms ± 18 µs/loop
Monte Carlo Pi	GPU	1.05 ms ± 76.5 µs/loop

Table 4: Hypotenuse function on CPU accelerated/Python

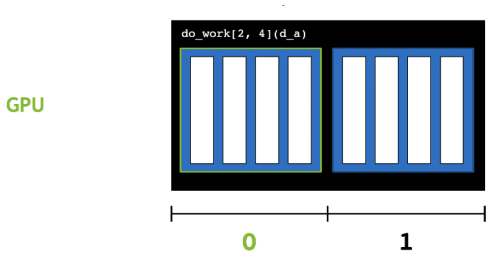

Figure 3: CUDA Thread Hierarchy Variables

ments. Numba provides the cuda.grid() function which returns a thread's unique index in the grid. The coordination of parallel threads has been very well represented in Figure 4.

Utilizing the concepts of grid stride loops we experiment with the Monte Carlo pi code. GPUs can be extremely useful for Monte Carlo applications where you need to use large amounts of random numbers. CUDA ships with an excellent set of random number generation algorithms in the cuRAND library. To test the Grid Stride Loop, we implement the Monte Carlo pi generation algorithm on GPU. The implementation time taken for the code on CPU and GPU has been shown in Table 4.

3. Model Details

In this section, we delve into the ablation studies conducted to unravel the intricate relationship between the model's size, number of parameters, running time, and accuracy. The study was guided by a key question: Does continuously increasing the model size invariably lead to an increase in accuracy? Through meticulous experimentation and hyperparameter tuning, this investigation aimed to answer this question. As an example, a machine translation task (English to Spanish) was taken, as it utilizes a complete encoder and decoder pipeline.

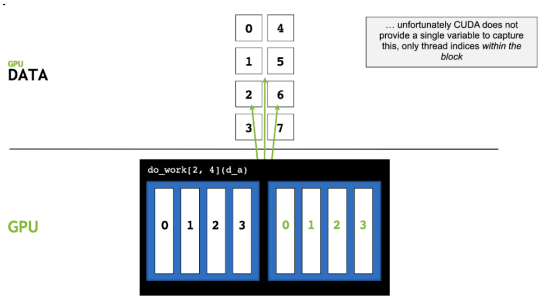

Figure 4: Coordinating Parallel Threads

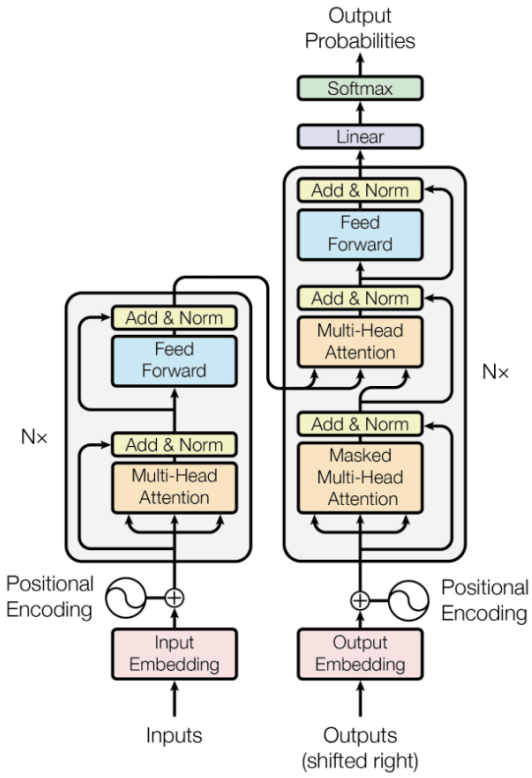

Figure 5: Transformer (Seq-to-Seq) Architecture

3.1. Data Preparation and Preprocessing

The experimental study focused on the machine translation task from English to Spanish. A dataset comprising 1 lakh translations was utilized, sourced from Kaggle. The chosen size of the problem was designed to experiment with the relationship between the model's size, number of parameters, and its performance.

3.2. Model Architecture

The Transformer model architecture as shown in Figure 5 was utilized in this study, renowned for its parallelization and the ability to capture long-range dependencies in the sequence data. The model consists of an encoder and decoder, each with multiple layers of attention and feed-forward neural networks. The following hyperparameters were explored: Model Size: Ranging from 16 to 512, reflecting the embedding dimensions. Number of Heads: Values varied from 4 to 16, influencing the model's attention mechanism. Number of Layers: Tested with 2 to 16 layers, impacting the depth of the network. Dropout Sizes: Ranging from 0.1 to 0.5, to prevent overfitting. Number of Epochs: Since this study was about evaluating the effect of an increase in the number of parameters on model accuracy, most of the cases were limited to 100 epochs. A few best cases were run till 400 epochs.

3.3. Training Configuration

The training was conducted on a single NVIDIA A100 GPU. The model's loss was computed using the cross-entropy loss

function for the machine translation task. A training-validation split of 70-30 per cent was employed.

4. Results and Discussion

4.1. Results with the model size of 16 on CPU

The results are shown in Figure 6 where we train the model on a CPU with a model size of 16, number of heads equal to 4 and 8, number of layers as 2 and 4 and a dropout of 0.1. We see the training accuracy, loss and perplexity for 100 epochs. Figure 7 curves represent accumulated time in minutes vs. epochs. 34 epochs of run in CPU. It took 1400 minutes in comparison to only around 120 minutes for GPU runs.

4.2. Results with the model size of 512

The results presented in Figure 9 reveal a disconcerting instability, particularly apparent when the model size is at its peak value of 512, with both the number of heads and layers set at 4. The instability in learning is suspected to be linked to a dropout value of 0.5, a feature typically employed to prevent overfitting through the introduction of noise. However, in this context, the chosen dropout value appears excessively high, inhibiting convergence and potentially suppressing essential features. This is exacerbated by the validation perplexity reaching into the millions, a stark and troubling indicator of the model's learning instability. With a reduction in the dropout value and the number of layers to 2, as depicted in Figure 10, the model begins to exhibit signs of learning, despite a persistently rugged learning surface. This intriguing pattern suggests an intricate interplay between the model's complexity (and thus parameter count) and the effectiveness of learning with a reduced dropout value. Figure 10 offers a closer examination of this phenomenon. Although learning begins to occur, convergence is not achieved. Validation losses and perplexity remain elevated even when the dropout is decreased to 0.3. An anomalous 'kink' in validation perplexity observed around the 100th epoch for a dropout of 0.4 (as illustrated in Figure 10(b)) may hint at the model's need for additional epochs to stabilize. This interpretation, though compelling, requires more rigorous testing or analysis to be validated.

4.3. Results with the model size of 256

Figure 11 presents a further reduction in model size to 256, exploring two specific combinations of heads and layers. For a configuration of 16 heads and 8 layers, the learning is minimal, denoted by a train accuracy improvement on the order of 1e-3. Conversely, the model demonstrates overfitting and becomes unstable, the ruggedness augmenting with increasing epochs. When heads are reduced to 4 and layers increased to 16, learning fails to occur, and perplexity surpasses 10 million for roughly 20 epochs. This indicates a mismatch in parameter selection or insufficient epoch count for learning to commence. Still, the enormity of perplexity suggests that more epochs may not salvage this configuration. In another set of experiments for a model size of 256 (Figure 12), the combination of 4 heads and 8 layers (with a dropout of 0.5) led to the model becoming unstable, starting to overfit, and displaying erratic perplexity. Other experimental combinations, such as 4 heads with 16 layers and a dropout of 0.5, showed overfitting; the setup with 8 heads, 16 layers, and a dropout of 0.5 was halted midway due to the onset of enormous perplexity scores and evident overfitting. With a reduction in the number of layers (Figure 13), the model began to show improvements, as evidenced by increasing validation accuracy and decreasing loss and perplexity. Notably, the perplexity scores were not as pronounced as in Figures 11 and 12. It was observed that for a model size of 256, with 4 heads and 2 layers, the reduction in dropout led to improved scores. This implies that a high dropout is not always favourable, and a lower dropout seemed to smooth the learning curve.

4.4. Results with the model size of 128

We further dropped the model size to 128. In Figure 14, the only configuration that showed signs of learning was with 8 heads and 4 layers, combined with a dropout of 0.5. However, near the end of the 98th epoch, this setup suddenly became unstable, with training accuracy plunging to 0.06 from 0.14. This demonstrated that none of the three combinations above were effective. The situation improved when the number of layers was reduced to 2 instead of 4 (Figure 15). This demonstrated that the model was learning with a lesser number of layers. From the validation accuracy and loss, it was observed that with a reduced number of heads in each layer (4 instead of 8), learning was more effective, and this progress continued up to 100 epochs.

4.5. Results with the model size of 64

In Figure 16, we further reduced the model size to 64 (from 128), keeping the number of heads, layers, and dropout value the same as in Figure 15. We observed a peculiar trend where the model learned for 100 epochs, then regressed, and eventually started overfitting, as training loss reduced while validation loss increased. This curious trend prompted us to attempt training for more epochs in select cases. In Figure 17, with the model size kept at 64 and the number of heads increased to 4, 8, and 16 (and layers increased to 8 with a dropout of 0.5), learning became unstable. Interestingly, when the number of heads was 16 and the layers were 4, the model performed better than other cases in Figure 17. This indicates that there might be a scope for improvement if trained for more epochs. From Figure 18, for a model size of 64, with 4 heads and either 2 or 4 layers (and dropouts of 0.3 and 0.4), the model started learning without any signs of overfitting. It is quite evident from this figure that increasing the number of encoder-decoder layers reduced the learning rate, even with a higher dropout value of 0.4.

4.6. Results with the model size of 32

Next, we reduced the model size further to 32, keeping the layers at 2 and the dropout at 0.5, but varying the number of heads (4 and 8). With a model size of 32, 4 heads, 2 layers, and a dropout of 0.5, the model began overfitting within 100

epochs. When we increased the number of heads to 8 (for a model size of 32 and 2 layers), intriguingly, the model did not overfit. However, the validation accuracy was still higher, and the validation loss was lower than in the previous case. This could simply be attributed to the increase in heads, and it's possible that the model would have started overfitting after 100 epochs. Next, we wanted to observe the effect of reducing the dropout from 0.5 to 0.1 (Figure 20). To our amazement, the model performed better in learning compared to the cases discussed in Figure 19. We tested a model size of 32, head sizes of 4 and 8, and a number of layers of 2 and 4, and found that the learning was comparable in all these cases. This suggested that with lower dropout, changes in the model head or layers might not significantly affect learning.

4.7. Results with the model size of 16

In our subsequent experiment, we reduced the model size further to 16 (Figure 21), and again the model was tested with 4 and 8 heads and 2 and 4 layers. In the case where the dropout was kept at 0.5, the learning was least effective, even with 8 heads for 2 layers. However, other cases with dropouts of 0.1 (and 4 or 8 heads with 2 layers) led us to rethink our previous approach of using higher dropouts like 0.5, 0.4, or 0.3. Overall, we observed that smaller dropout values yielded better results.

4.8. Results with increasing reducing dropout value to 0.1

Next, to test our hypothesis that a reduction in dropout value yields more stable results (i.e., from 0.5 to 0.1), we experimented with larger model sizes (128, 256) but kept the number of heads fixed at 4 and used only smaller encoder- decoder layers (2, 4). We observed that the model's accuracy was highest within 100 epochs for the configuration with a model size of 128, heads and layers equal to 4, and it was still learning even after 100 epochs. Increasing the model size to 256 reduced the validation accuracy and increased the loss, and changing the dropout from 0.1 to 0.5 further degraded learning. As evidenced by the Figure 8 table, it took just 26 million parameters (model size: 128, heads: 4, layers: 4) to achieve the best performance, and performance also degraded with an increase in the number of parameters.

4.9. Best Results from Table in Figure 8

A comprehensive review of the table reveals an intriguing pattern that runs contrary to common expectations in model design; namely, that an increase in the number of parameters does not consistently correspond to a decrease in validation perplexity. This is vividly illustrated in the comparison between configurations with different model sizes, numbers of heads, layers, and dropout rates. Remarkably, the combination of a model size of 128, 4 heads, 4 layers, and a dropout rate of 0.1 achieved the best performance with just 26 million parameters. Even configurations with higher complexities and more parameters, such as those with 256 or 512 model sizes, failed to surpass this level of efficiency. Validation perplexity in some of these larger models escalated to alarming figures, as evidenced in specific rows of the table (Table in Figure 8)

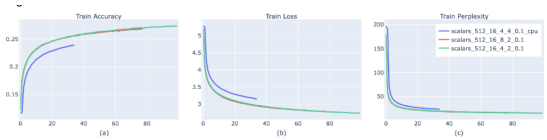

Figure 6: Experiments: Blue curve represents cpu run (34 epochs). Maximum length = 512, Model Size = 16, number of heads = 4, 8, number of layers = 2, 4 and dropout sizes = 0.1 (combinations shown in legends). Ran for 100 epochs (x-axis). **(a)** Train Accuracy. **(b)** Train Loss. **(c)** Train Perplexity.

4.10. Investigating Best Results from Table in Figure 8 further by reducing dropout to 0

Lastly, we picked up the best configurations, according to the table in Figure 8, and we increased the number of epochs (Figure 23). This time we extended the runs between 350-400. In the best configuration (with model size of 128), we decrease the dropout to 0, to test the effect. We also reduced the model size to 64 and the dropout to 0, and we did one more test with the model size of 512. These tests were done to capture the effects on model performance with lower/zero dropouts, higher number of epochs, and range of parameters. We observed that decreasing the dropout to 0 in model size of 64 and 128 led to overfitting showing that for better performance dropouts are needed. Similarly, We observed that when the model size is bigger (model size of 512), it is still overfitted even with a dropout of 0.1. Perhaps, the higher dropout would have taken care of the overfitting, however, its accuracy was way less than the model size of 512. The investigation into various configurations of model sizes, numbers of heads, layers, and dropout values has unveiled a complex and multifaceted relationship between model complexity and learning efficacy. Contrary to conventional wisdom, the results consistently demonstrate that increased model complexity, as characterized by larger model sizes and more heads and layers, does not necessarily yield improved learning performance. In fact, certain simpler configurations, notably a model size of 128 with 4 heads and 4 layers and a lower dropout rate of 0.1, outperformed more complex structures, achieving superior performance with just 26 million parameters. This unexpected outcome emphasizes that a naive escalation in complexity and parameter count can not only inhibit learning but may lead to instability and overfitting. In a stark departure from the common pursuit of ever-larger models, these findings advocate for a more nuanced understanding of model architecture and hyperparameter tuning, balancing complexity against efficiency and stability.

5. Summary and conclusions

- Python Accelerated: We studied the various components of Python accelerated multiprocessing. While using CUDA might seem like a great option, it is not the way to go for smaller problems that give us an overhead which isn't worth the wait. In those cases, in-built Python versions might work the best. Checking what suits best for our function is how we can optimize its performance.

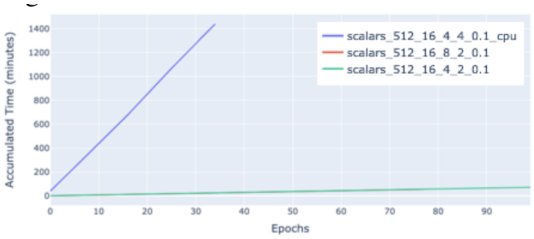

Figure 7: Experiments: Blue curve represents cpu run (34 epochs). Curves represent accumulated time in minutes vs. epochs. 34 epochs of run in CPU took 1400 minutes in comparison to only around 120 minutes for GPU runs.

Model	Heads	Layers	Dropout	Time (min)	Train Loss	Val Loss	Val Perplexity	Parameters (million)
16	4	2	0.1	71.0	2.7389	2.3684	10.6806	3.20
16	4	4	0.1	93.0	2.9223	2.5788	13.1816	3.21
16	8	2	0.5	70.0	4.3441	4.477	87.9667	3.20
16	4	2	0.5	71.0	4.4678	6.1332	460.9081	3.20
32	4	4	0.1	98.0	1.8019	1.5429	4.6783	6.42
32	4	2	0.1	67.0	1.669	1.4727	4.361	6.36
32	4	2	0.5	67.0	3.7191	10.8591	5.2e+04	6.36
16	4	2	0.5	71.0	4.4678	6.1332	460.9081	3.20
32	8	2	0.5	68.0	3.6843	5.202	181.6349	6.36
64	4	8	0.5	244.0	5.2703	8.035	3.1e+03	13.48
64	8	8	0.5	176.0	5.0072	9.399	1.2e+04	13.48
64	16	4	0.5	98.0	4.5781	6.232	508.7538	13.01
64	4	2	0.4	71.0	2.2984	1.9782	7.23	12.78
64	4	2	0.3	71.0	1.7956	1.5268	4.6035	12.78
64	4	4	0.4	94.0	3.1409	3.3198	27.6538	13.01
64	4	4	0.3	92.0	2.0916	1.7769	5.9116	13.01
64	4	2	0.5	121.0	2.9436	7.4402	1.7e+03	12.78
128	4	2	0.1	79.0	0.5859	0.9217	2.5136	25.95
128	**4**	**4**	**0.1**	**107.0**	**0.6481**	**0.8993**	**2.4579**	**26.87**
128	8	2	0.5	79.0	2.4338	2.9636	19.3666	25.95
128	8	4	0.5	110.0	5.062	11.2901	8.0e+04	26.87
128	4	8	0.5	178.0	5.2745	11.5297	1.0e+05	28.73
128	4	4	0.5	106.0	5.2141	12.1386	1.9e+05	26.87
256	4	2	0.1	94.0	0.3654	1.0448	2.8427	53.67
256	4	4	0.1	120.0	1.4242	1.9795	7.2392	57.36
256	4	2	0.4	94.0	2.036	2.5577	12.906	53.67
256	4	2	0.3	93.0	1.6566	1.8869	6.599	53.67
256	4	2	0.5	94.0	2.2395	2.9339	18.8001	53.67
256	16	8	0.5	209.0	5.2948	11.2313	7.5e+04	64.73
256	4	16	0.5	314.0	5.2841	12.7906	3.6e+05	79.47
256	4	16	0.5	314.0	5.2841	12.7906	3.6e+05	79.47
512	4	4	0.5	166.0	5.3473	14.5356	2.1e+06	129.34
512	4	2	0.5	229.0	2.9624	3.4459	31.371	114.62
512	4	2	0.4	135.0	2.6893	3.4579	31.7515	114.62
512	4	2	0.3	133.0	2.5512	2.7891	16.2657	114.62

Figure 8: An analysis of the impact of different hyperparameters on machine translation model performance, along with the corresponding parameter count for each configuration (for 100 epochs)

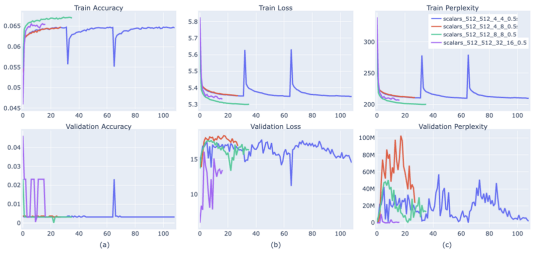

Figure 9: Experiments: Maximum length = 512, Model Size = 512, number of heads = 4, 8, 32, number of layers = 4, 8, 16, and dropout sizes = 0.5 (combinations shown in legends). (a) Train and Validation Accuracy: Comparison of training and validation accuracy across different configurations. (b) Train and Validation Loss: Plot of training and validation loss for different models. (c) Train and Validation Perplexity: Visualization of training and validation perplexity under various hyperparameter settings.

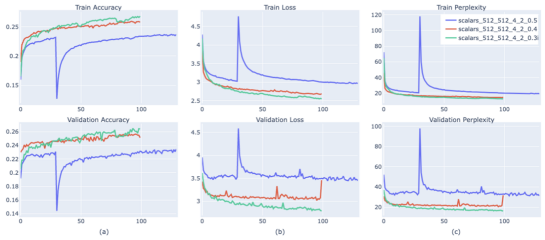

Figure 10: Experiments: Maximum length = 512, Model Size = 512, number of heads = 4, number of layers = 2, and dropout sizes = 0.3, 0.4, 0.5 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

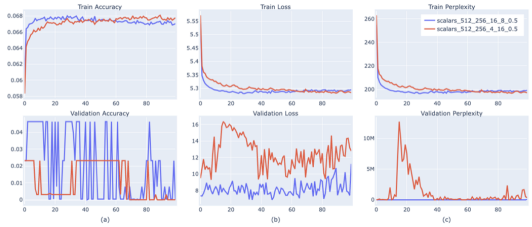

Figure 11: Experiments: Maximum length = 512, Model Size = 256, number of heads = 4, 16, number of layers = 8,16, and dropout sizes = 0.5 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

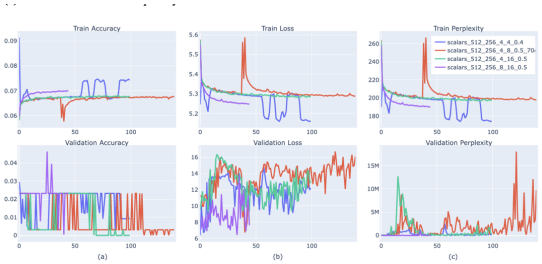

Figure 12: Experiments: Maximum length = 512, Model Size = 256, number of heads = 4, 8, number of layers = 4, 8, 16, and dropout sizes = 0.4, 0.5 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

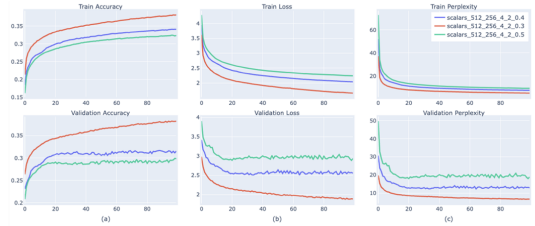

Figure 13: Experiments: Maximum length = 512, Model Size = 256, number of heads = 4, number of layers = 2, and dropout sizes = 0.3, 0.4, 0.5 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

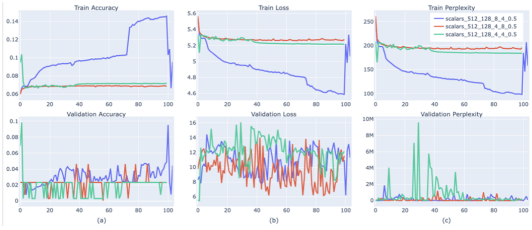

Figure 14: Experiments: Maximum length = 512, Model Size = 128, number of heads = 4, 8, number of layers = 4, 8, and dropout sizes = 0.5 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

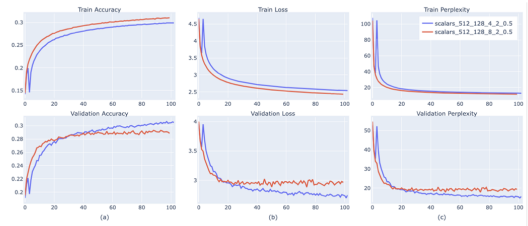

Figure 15: Experiments: Maximum length = 512, Model Size = 128, number of heads = 4, 8, number of layers = 2, and dropout sizes = 0.5 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

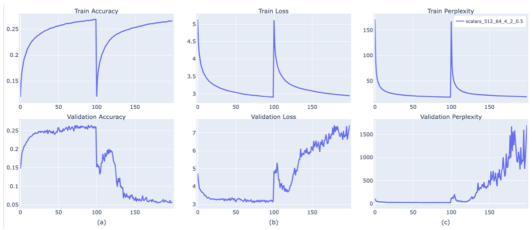

Figure 16: Experiments: Maximum length = 512, Model Size = 64, number of heads = 4, number of layers = 2, and dropout sizes = 0.5 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

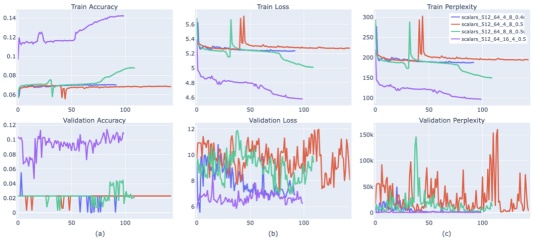

Figure 17: Experiments: Maximum length = 512, Model Size = 64, number of heads = 4, 8, 16, number of layers = 4, 8, and dropout sizes = 0.5 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

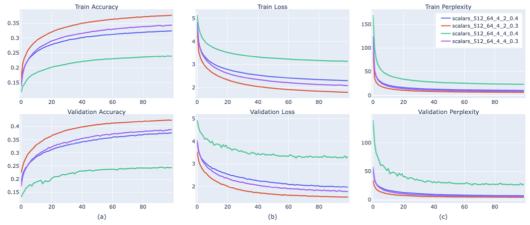

Figure 18: Experiments: Maximum length = 512, Model Size = 64, number of heads = 4, number of layers = 2, 4, and dropout sizes = 0.3, 0.4 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

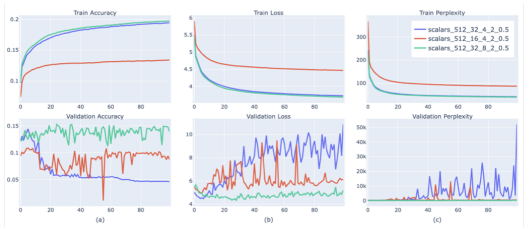

Figure 19: Experiments: Maximum length = 512, Model Size = 32, 16, number of heads = 4, 8, number of layers = 2, and dropout sizes = 0.5 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

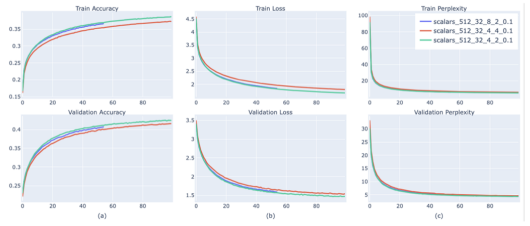

Figure 20: Experiments: Maximum length = 512, Model Size = 32, number of heads = 4, 8, number of layers = 2, 4 and dropout sizes = 0.1 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

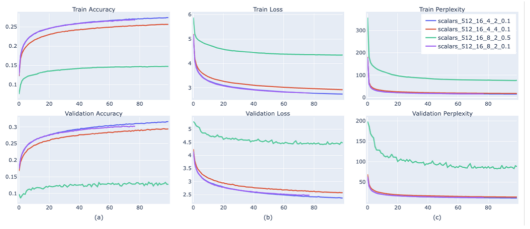

Figure 21: Experiments: Maximum length = 512, Model Size = 16, number of heads = 4, 8, number of layers = 2, 4 and dropout sizes = 0.1, 0.5 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

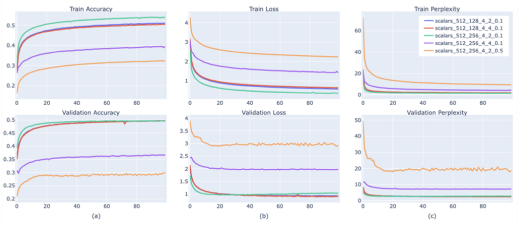

Figure 22: Experiments: Maximum length = 512, Model Size = 128, 256, number of heads = 4, number of layers = 2, 4 and dropout sizes = 0.1, 0.5 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

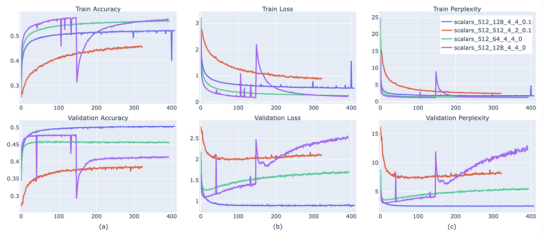

Figure 23: Experiments: Maximum length = 512, Model Size = 64,128, 512, number of heads = 4, number of layers = 2, 4 and dropout sizes = 0.1, 0 (combinations shown in legends). (a) Train and Validation Accuracy. (b) Train and Validation Loss. (c) Train and Validation Perplexity.

- Model Complexity vs. Efficiency: The study's findings highlight that increasing model size and complexity does not necessarily lead to better learning performance. Moderate complexity in configurations at times outperformed larger, more intricate setups.
- Avoidance of Excessive Compute Power: Rather than simply throwing more compute power and multiple GPUs at the problem, the study emphasizes a nuanced relationship between complexity and efficiency. It revealed that this approach could lead to instability, overfitting, and counterproductive results.
- Role of Dropout: The analysis unveiled the multifaceted role of dropouts in learning. The fine-tuning of this hyperparameter was shown to be vital, with too high or too low values leading to hindrances in convergence and learning.
- Overfitting Trends: A balanced choice of model size, heads, layers, and dropout was found to be crucial for optimal learning without overfitting. Thoughtful hyperparameter tuning is preferred over a brute force increase in complexity.
- Influence of Heads and Layers: These parameters had a nuanced impact on the learning process, illustrating the importance of understanding the complex interplay between different hyperparameters.
- Evaluation Metrics and Perplexity: The study made significant use of validation loss curves and perplexity as indicators of learning stability and efficiency, focusing on these over traditional metrics like BLEU scores to understand overfitting and the impact of parameter increases on accuracy.
- Advocacy for Thoughtful Tuning: Some simpler configurations achieved outstanding performance with only 26 million parameters. This demonstrates the importance of an intelligent, balanced approach that considers various hyperparameters instead of merely increasing resources and model sizes.
- Counterintuitive Findings: The results challenged conventional wisdom, notably achieving the best per- formance with a configuration that was neither the largest nor most complex. It further emphasizes that optimization can often achieve better outcomes without relying on significant computational assets.
- Need for Hyperparameter Optimization: Overall, the study underscores the importance of meticulous hyperparameter tuning. It acts as a reminder that less can be more if paired with an understanding of the intricacies of model architecture and careful optimization.

References

[1] NVIDIA Deep Learning Institute Title of the course https://courses.nvidia.com/courses/course-v1: DLI+C-AC-02+V1/, 2019

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017

[3] NVIDIA Corporation. NVIDIA A100 GPU Architecture. https://www.nvidia.com/en-us/data-center/ a100/, 2020

[4] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. arXiv preprint arXiv:2203.15556, 2022

[5] TensorFlow. Spanish-English translation corpus http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip, 2022