A THE SLURM QUEUEING SYSTEM

Proposed in 2003, the SLURM (Simple Linux Utility for Resource Management) task manager is a framework for multi-user hierarchical control [22], which is now widely accepted by companies and academia. As super-computers are often used by multiple users in an organization, Linux needs to know how to schedule all the tasks received and make a relatively rule for each user to share computational resources.

When the code and data are in place, the user is required to write a script to launch the task, including which nodes to use, how many GPUs are needed, and the maximum time for executing the script. The script is called by `sbatch <script>`, so it becomes the entry point of the whole program. Essentially, it's just a `bash` script indicating resources required of the task, and what the task does. The script will be executed in a blocking way, so the second command in the script must wait until the first command is completed and returned.

As SLURM assigns the user tasks randomly, it's hard for the user to know which node he is allocating before he launches the script. This problem can be solved by requesting a reservation on SLURM.

B MULTI-HEAD ATTENTION LAYER

The multi-head attention layer is the crucial concept proposed by the transformer architecture to parallelize temporal data modeling. As this layer is crucial in the GPT-2 model, we give a mathematical introduction here for readers to refer to.

B.1 Variances Of Recurrent Neural Network

Before the transformer was proposed, the most popular sequential models were all done by the Recurrent Neural Network (RNN). The basic ideology of each RNN layer is to input a sequence of vectors (words) and output a sequence of vectors with the same position and length. In other words, each RNN layer turns a sentence into another sentence with the same length. Because RNN cannot get future information, bidirectional RNN is proposed, as shown in Figure 10.

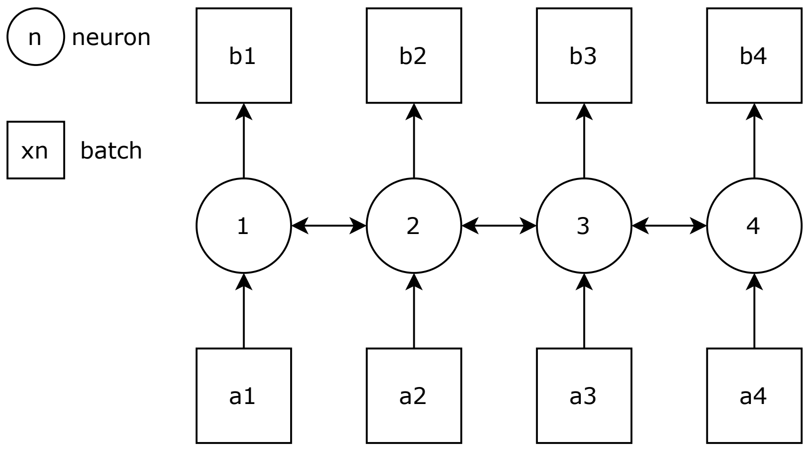


Fig. 10. The architecture of Bidirectional RNN.

As indicated by its name, the communication among the neurons is bidirectional, which means a neuron can get information from both sides of a word in the sentence. However, RNN cannot be parallelized. For example, to get the result b_4, we have to first calculate b_1 to b_3. To solve this problem, the idea of **Convolutional RNN** is proposed: CNN be used to replace RNN, as shown in Figure 11.

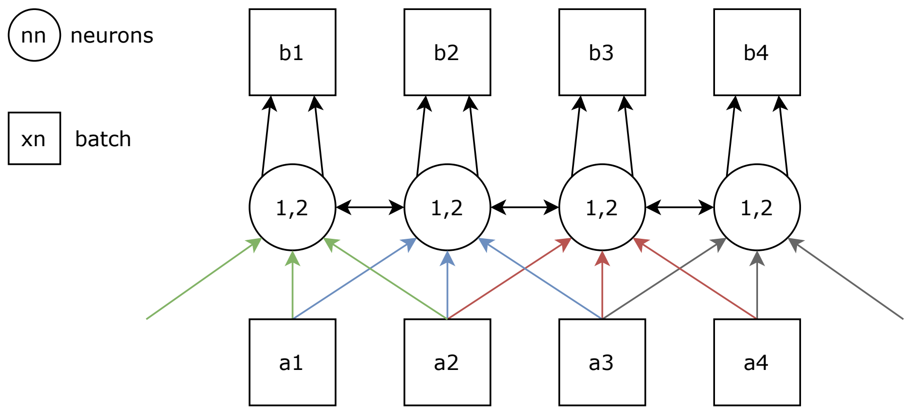


Fig. 11. The architecture of Convolution RNN.

In the figure, each CNN neuron takes 3 inputs: the current word, the last word, and the next word. Since the output of each neuron is only a scalar, we use multiple neurons to represent the whole vector. For example, if the input vector has a dimension of 2, or $len(a_i) = 2$, we need 2 neurons to produce 2 scalars to construct the vector. This architecture is much easier for parallelization.

However, CRNN is still not satisfying, because each CNN neuron can only consider a very limited amount of information. In the last example, we can only detect 3 vectors for one CNN neuron. We call the number of detectable vectors Receptive Field. This leads to much worse performance for CNN compared to RNN, as RNN makes use of all the information from previous neurons or even the whole sentence.

An attempt to solve this problem is a hierarchical design of the CNN layers, as shown in Figure 12. However, this model still does not improve the receptive field for the first neuron.

B.2 The Self-Attention Layer

To solve this problem completely, the idea of the self-attention layer was introduced. The task of the self-attention layer is exactly the same as RNN - it takes a sentence as input, and outputs a sentence of the same size. The difference is that all the words are calculated and output at the same time, so they can be paralleled and easier for GPUs to process. Simply put, the self-attention layer is a paralleled RNN.

An example of the architecture of the self-attention layer is shown in Figure 13. The input is three words labeled x_i, and each of them goes through an embedding function to get its word vector a_i. Then the model defines 3 matrices, which are called the attention heads: the Q matrix for query, the K matrix for key, and the V matrix for weighted word vector. Practically, the query is used to match all keys to produce the attention vector α_{ij}, which is represented as "at" in the figure. We can represent the calculations in Formula 1.

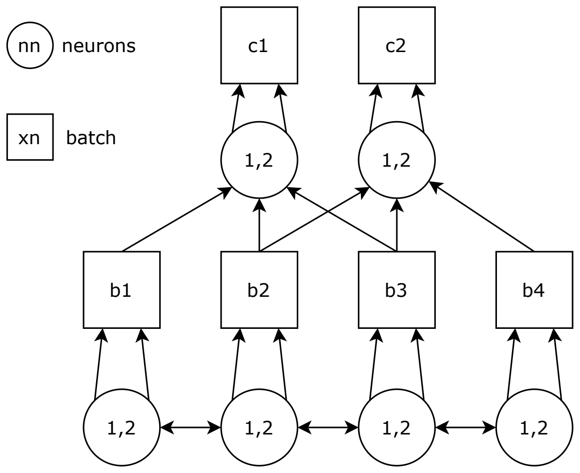


Fig. 12. The architecture of the Hierarchical CRNN.

$$\alpha_{ij} = \frac{q_i \cdot k_j}{\sqrt{d}} \tag{1}$$

$$q_i = Qa_i \tag{2}$$

$$k_i = Ka_i \tag{3}$$

$$v_i = Va_i \tag{4}$$

The α_i attention vectors are then normalized by passing through the soft-max layer and become the standardized attention vectors s_i. This vector does a dot-product with the weighted word vector v_j and gains the weight vector w_j for this word. This is why v_i is called the weighted word vector - it is directly extracted from the embedded word and be summed up with the other vector, thus gaining the weight of this word. The word vector can represent the information stored in the word very well.

At last, we sum up the three weight vectors and get the final result b_1. Repeating this process 3 times, we can get the corresponding output vector b_2, b_3 for a_2, a_3 too, and thus outputting exactly the same form of data like RNN - one b_i for each a_i.

Looking back at the gained vector b_i, we surprisedly find that it has already taken all information in the input sentence A. Consider the case we have a sentence of length $l > 3$, the parallization will be much more apparent and thus b_1 has a larger receptive field with a longer sentence.

However, for the naive model shown above, it's still not obvious how everything can be parallelized. For example, we have to get s_{11} to get w_{11}. In Section B.3, we dicuss how to parallelize the calculation using linear algebra (matrix).

B.3 Parallization Of The Self-Attention Layer

Firstly, we notice that the transformer matrices Q, K, V (we now represent them as W^Q, W^K, W^V as they're essentially weights) for each word are the same. Thus, we can use one matrix to calculate all the results in one-shot using the

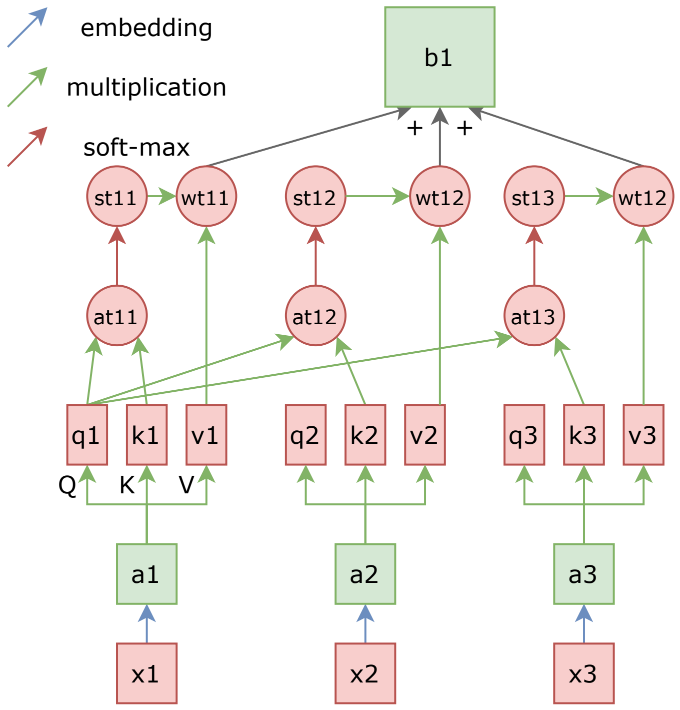


Fig. 13. The architecture of a self-attention layer.

formula $[q_1 q_2 q_3 q_4] = W^Q[a_1 a_2 a_3 a_4]$, where $[q_1 q_2 q_3 q_4]$ is the matrix constructed by stacking the four query vectors. We apply the same strategy to W^K and W_V for parallelization.

$$Q = W^Q A, K = W^K A, V = W^V A \tag{5}$$

$$Q = [q_i]_1^l, K = [k_i]_1^l, V = [v_i]_1^l \tag{6}$$

$$A = [a_i]_1^l \tag{7}$$

Secondly, we observe that the calculation of the attention vectors α_i can be parallelized because they're all using the same query vector q_i. Thus, we construct the attention matrix $\mathcal{A}$ as shown below.

$$\begin{bmatrix}\alpha_{11}\\ \alpha_{12}\\ \alpha_{13}\end{bmatrix} = \begin{bmatrix}k_1\\ k_2\\ k_3\end{bmatrix} \cdot q_1 \tag{8}$$

We can apply the same process to q_2 and q_3, and we find that the Q matrix can be used to solve the whole problem.

$$\begin{bmatrix} \alpha_{11} & \alpha_{21} & \alpha_{31} \\ \alpha_{12} & \alpha_{22} & \alpha_{32} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} = \begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} \cdot \begin{bmatrix} q_1 & q_2 & q_3 \end{bmatrix} \tag{9}$$

If we generalize the matrix, we get

$$\mathcal{A} = ([k_i]_1^l)^T \cdot [q_i]_1^l = K^T Q \tag{10}$$

Then we apply the softmax function and get

$$\hat{\mathcal{A}} = Sm(\mathcal{A}) \tag{11}$$

At last, we can parallelize the last step as well. Taking our example as the starting point, we have

$$b_1 = [v_1 v_2 v_3] \cdot \begin{bmatrix} \hat{\alpha}_1 \\ \hat{\alpha}_2 \\ \hat{\alpha}_3 \end{bmatrix} \tag{12}$$

$$b_1 b_2 b_3 = [v_1 v_2 v_3] \cdot \begin{bmatrix} \hat{\alpha}_{11} & \hat{\alpha}_{21} & \hat{\alpha}_{31} \\ \hat{\alpha}_{12} & \hat{\alpha}_{22} & \hat{\alpha}_{32} \\ \hat{\alpha}_{13} & \hat{\alpha}_{23} & \hat{\alpha}_{33} \end{bmatrix} \tag{13}$$

To put in matrix, we have the simplified form

$$B = V \cdot \hat{\mathcal{A}} \tag{14}$$

Now we make a conclusion of the task done by the self-attention layer using matrix. First, we get the intermediate matrices - the query matrix, the key matrix, and the vector matrix. Then we get the attention matrix using the key matrix and the queue matrix. After that we apply the soft-max function on the attention matrix to get the standardized attention matrix. At last, we use the vector matrix and the standardized attention matrix to get the output matrix B. The whole process is shown below.

$$Q = W^Q A, K = W^K A, V = W^V A \tag{15}$$

$$\mathcal{A} = K^T Q \tag{16}$$

$$\hat{\mathcal{A}} = Sm(\mathcal{A}) \tag{17}$$

$$B = V\hat{\mathcal{A}} \tag{18}$$

B.4 Multi-head Attention

The multi-head attention splits the attention head up to multiple smaller heads, and the architecture is shown in Figure 14. Using the multi-head attention layer, the dimension of the features is further expanded to a factor of the number of attention heads.

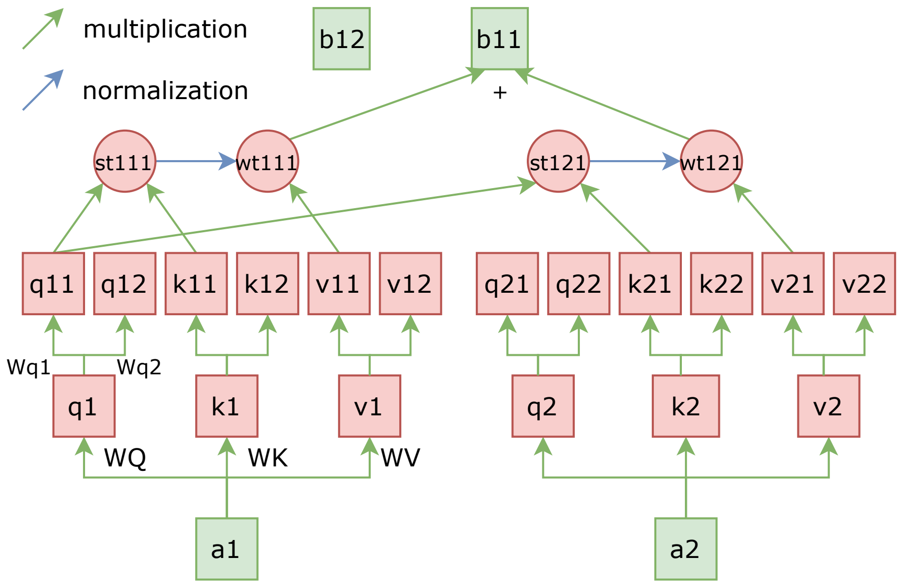


Fig. 14. The architecture of a multi-head attention layer.

For each attention head query q_i, we use matrices to further split it. In the example in the figure, we have 2 multihead attention matrices. Taking the query matrix as example, the formula is shown below.

$$q_{ij} = q_i W_{qj} \tag{19}$$

This process happens exactly the same for vectors k and v. In the example above, now we get all the attention heads q_{im}, k_{im}, v_{im}, where m represents the number of multi-head heads. Then we use the splitted attention head q_{11} to calculate each attention with k_{11}, k_{21}, and thus get $\alpha_{111}, \alpha_{121}$ and w_{111}, w_{121}. Finally, we dot product the two weight vectors and get b_{11}. After getting all the b_{im} results, we can stack them and multiply by an output matrix W_b, as shown below.

$$b_i = W_b \begin{bmatrix} b_{i1} \\ \dots \\ b_{im} \end{bmatrix} \tag{20}$$

This dimension expansion strategy is used to detect more information in the system. For example, if we split one head into 2, with each head multiplied by a weight matrix, we can train the parameters so that one split head is used for detecting the information of the neighbor words, and the other is used for detecting the global information.

B.5 Positional Encoding

As mentioned in Section B.2, the self-attention layer takes all information in the sentence to train one word. Thus, this word has no information about where the words it's learning from are in the sentence. This can become a serious problem, as failure to locate words makes the transformer lose the ability that RNN has. To overcome it, the Positional Encoding technique is introduced, which adds a position vector e_i to the embedded word a_i.

This process has a prototype of concatenation. Suppose we have a one-hot positional vector p_i, in which the only number 1 indicates the position of the word in a sentence, then we have the formula:

$$\begin{bmatrix} W_A & W_P \end{bmatrix} \begin{bmatrix} x_i \\ p_i \end{bmatrix} = W_A \cdot x_i + W_P \cdot p_i = a_i + e_i \tag{21}$$

C INSIGHT OF GPU MEMORY USAGE

In our research, we encountered lots of GPU Out Of Memory (OOM) issues, and we managed to propose several possible solutions to it by investigating the essence of the error. In Appendix C, we're going to analyze the GPU usage in the training process, and in Appendix D we're going to solve the OOM errors and discuss why the errors were encountered in our research.

C.1 The CUDA GPU Memory

GPU memory is different from CPU memory because GPU memory is exclusively used by the GPU streaming multiprocessors on the card, while CPU memory (which is often called the general memory or simply “memory") is used by a series of devices including the CPU itself. The bandwidth of GPU memory (GDDR6) is 3 times faster than the general memory (GDDR4) due to a great requirement of high-performance computing, and thus it's not recommended to train a deep neural network on a CPU instead of a GPU.

However, as CPU memory is commercialized and can be installed by the user himself, the size of CPU memory is often much larger than the GPU memory. Thus, under extreme circumstances[13], it's still a valid way to change GPU memory to CPU memory to get the OOM issue solved, as will be discussed in Appendix D.5.

C.2 Estimating the Mamory Use of Models

Before solving the OOM issue, we need to first estimate which part of the model causes the usage of CUDA memory [23]. Basically, it's divided into three parts: parameters of the model, parameters of the optimizer for the model, and I/O for each layer in the model.

C.2.1 Parameters of the Layer (p_l). The number of parameters of the model is simply the sum of the weight and bias in all layers in the model, and doesn't depend on the size of input data. In detail, the model is loaded after the command `model = MyGreatModel(). cuda()` or `model.to(cuda:{i})`, and can be examined using the command `model.parameters()`. It should also be noted that only some layers have weights (like CNN layer, RNN layer, FCN layer, BatchNorm layer, and Embedding Layer), while the others do not (like activation layer, dropout layer, and pooling layer) [9]. Table 6 is a summary of the layers with their layer weights amount.

[13]Extreme circumstance means cases where no multi-GPU solution is possible, a large amount of memory is required to start the training (more than one GPU can satisfy), and cannot solve by reducing the model size.

Layer	Layer Input	Layer Output	Weights Amount p_l
Linear	m	n	$m \cdot n$
Conv2d	c_i, k	c_o	$c_i \cdot c_o \cdot k^2$
BatchNorm2d	n, c, h, w	n, c, h, w	$2 \cdot n$
Embedding	n, w	n, w, h	$n \cdot w$

Table 6. Parameter number in memory for one layer in PyTorch, originally proposed by [21].

Note that the number of parameters is not the same as the memory usage in memory, as each parameter has a data size. For example, `float32` will multiply the number argument by 4 to get the memory usage, `double` will multiply it by 8, and so on.

C.2.2 Parameters of the Layer After Optimization (p_r). The parameter of the optimizer is the parameter generalized when back-propagating in the optimization state. In other words, the parameter of the optimizer is the gradient ($\Delta F(w)$) and momentum (v).

As different optimizers utilize different optimization technologies, the actual amount of the parameters of the optimizer is also variable. For `SGD`, it takes approximately the same parameters as the model, which means that the parameter amount will be doubled after this optimizer. For `Adam`, it takes approximately 3 times as much as the model parameter, including the size of SGD, and double the size of momentum information, which means that the total parameters will be 4-fold after optimization. For reference, the memory storage of the optimizers is shown in Table 7 [3].

Layer Optimizer	Layer	Gradient	Momentum	Result p_r	Factor n
SGD	p_l	p_l	0	$2p_l$	2
SGD-Momentum	p_l	p_l	p_l	$3p_l$	3
Adam	p_l	p_l	$2p_l$	$4p_l$	4

Table 7. Parameter number in memory for one layer after applying the optimizer in PyTorch, originally proposed by [23].

In PyTorch, the optimizer is not layer-specific, as each optimizer applies to all layers in a model. Thus, we first calculate the model parameter without the optimizer and then multiply it by the optimizer factor. If we represent $p_r^{l_i}$ as the parameter in layer i, l_i as the layer number, and p_m as the parameter in the whole model, then the parameter number in the whole model can be calculated by Formula 22, as shown below.

$$p_m = \sum_{i=0}^{n} p_l^{l_i} \cdot n \tag{22}$$

C.3 Layer Output and Model Input

Except for memory for each layer, we also need memory for data produced by each layer. Thus, if we represent $p_o^{l_i}$ as the sum of all the outputs of layer i, the total memory usage for storing all the parameters for each layer output can be calculated by Formula 23. This formula holds because the output of each layer is the input of the next layer - we don't need to add the layer inputs in the formula.

$$p_o = \sum_{i=0}^{n} p_o^{l_i} \tag{23}$$

The size of input data for the model can usually be omitted, because we always use an iterator to split the data into batches - a 5-GB large dataset can be split up into a large amount of 1-MB data, which is handled by the data loader. The illustration of the layer output and model input contributing to memory storage is in Figure 15.

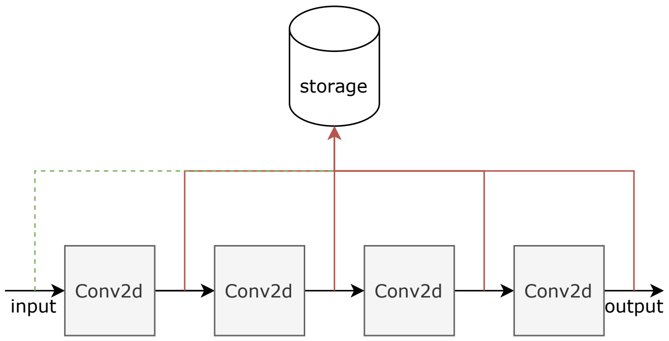


Fig. 15. The illustration of storage of output of each layer p_o (the read lines) and the storage of input of the model p_b (the dashed green line). The latter one is usually ignored because we typically don't calculate it's gradient, and it's usually split into pieces so it does not occupy too much storage.

For the question about p_o in the case that the input data is large: although the input data of the model p_b is also multiplied by a large number, causing a large memory usage, it is still relatively small than p_o.

C.4 Concluded Formula

Taking all 3 ingredients into consideration, we have the CUDA memory calculation Formula 24, as shown below.

$$\mathcal{M} = p_m \cdot n + b \cdot p_o + p_b \tag{24}$$

$$p_m * n = \sum_{i=0}^{n} p_r^{l_i} \tag{25}$$

Here $\mathcal{M}$ is the CUDA memory usage, p_m is the parameter of the model, p_o is the sum of parameters of the output for each layer, p_b is the size of the input data for the whole model, b is the batch size, and n is an integer usually between 2 and 6, which is subjective to the type of optimizers, like 2 for SGD and 4 for Adam. Note that the last item (p_b) can usually be ignored as discussed in Section C.3.

D REDUCING GPU MEMORY USAGE

In this appendix, we illustrated the practical ways to reduce memory usage and the reason for the unexpected OOM issues in our experiments, as illustrated in Section 4.2. In this part, we frequently refer to Formula 24 so it's recommended to go through Appendix C first.

D.1 Change Data Length using Apex

The most efficient strategy, also the easiest one, is to shift the processed data to another type with a shorter length. For example, if the data is now floating-point-32, consider shifting it to floating-point-16 for all data, including the model and the input data. This approach decreases a bit of the precision, but it decreases the usage of the memory to approximately a half [20].

This can be proved by Formula 24. Because all the parameter sizes (p_m, p_o, p_b) are decreased by a factor of 2, the whole usage $\mathcal{M}$ is also decreased by a factor of 2.

However, changing data types manually is time-consuming and bug-prone, and can also decrease the precision non-trivially. Thus, PyTorch developed a very powerful library utilizing NVIDIA Tensor Core called Apex. The Apex library takes floating-point-32 as original code but utilizes in-library optimizations to the data using a mixed distribution of floating-point-16 and floating-point-32 to decrease the usage of data to a factor of approximately 2. This strategy is proved to lose only a trivial precision when training and requires adding only a few lines into the original code with floating-point-32 data.

However, mixed-precision training using `fp16` (half precision) is now only available for GPUs supporting NVIDIA Tensor Core. Fortunately, the *HAL* system is built using V100 GPUs based on the Volta architecture, which has the built-in Tensor Core, so it's safe to use in our experiments.

The detailed code snippet for utilizing the Apex library can be found on the official website at https://github.com/NVIDIA/apex.

D.2 Minimize Batch Size

Minimizing batch size is another crucial method to reduce $\mathcal{M}$. According to Formula 24, when b is halved, the second item is halved. Thus, when the first item is small,[14] this is very useful for reducing the memory usage. Note that reducing both development batch size and testing batch size can contribute to a smaller $\mathcal{M}$ [21].

However, small batch size is very likely to exert a performance loss of the convergence of the loss curve. For example, in our GPT-2 training process, if the batch size becomes 1 or 2, the model will not be sufficiently fed for each step and thus create a tendency of divergence in the loss curve. We've done several experiments, with the results shown in Figure 16. The results show that a smaller batch size tends to make the loss curve not as smooth and prone to diverge. Even if we observe a normal loss curve, a small batch size may also lead to results that are not meaningful.

Thus, researchers should double-check the influence of reducing batch size in the model they're investigating, as the conclusion varies from model to model.

D.3 Reduce Input Sequence Length

Reducing the input sequence length is another popular way to tackle the OOM issue. Note that this method is more likely to be used in a sequential model that is order-sensitive, such as data used in RNN models. In these kinds of training, the sequence length of the data is linear to p_o, thus a halved sequence length has the same effect as halved batch size b, leading to approximately the same memory saving as discussed in Section D.2.

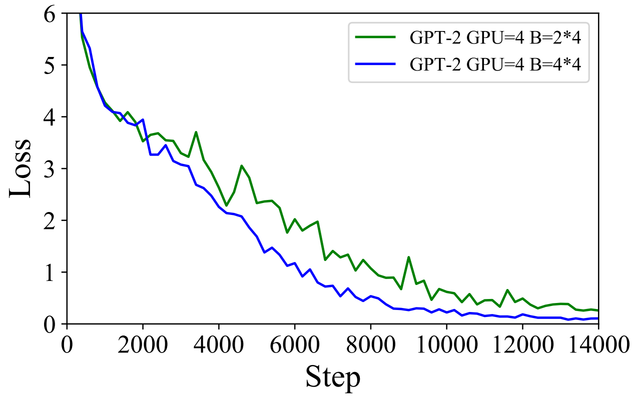
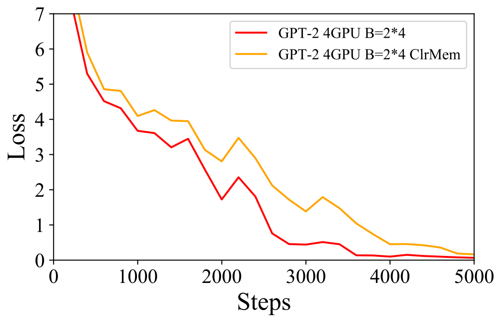
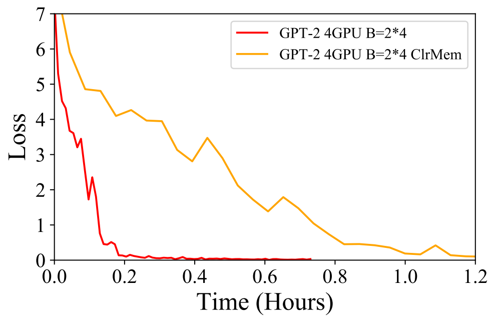


Fig. 16. The loss curve of running GPT-2 under 4 GPUs with batch size 2*4 and 4*4.



(a) The loss-step curve of running GPT-2 under 4 GPUs with batch size 2*4 with memory clear and without memory clear.



(b) The loss-time curve of running GPT-2 under 4 GPUs with batch size 2*4 with memory clear and without memory clear.

Fig. 17. The experimental results of the loss curve of GPT-2 under 4 GPUs before/after applying the memory clean code, with respect to number of training steps (a) and number of hours (b).

D.4 Free Unnecessary Variables

Compared to other strategies, this strategy is more engineering-based and has nothing to do with Formula 24. If we represent the real memory usage as $\mathcal{M}_r$, and the marginal cost by the code itself as $\mathcal{M}_c$, we have $\mathcal{M}_r = \mathcal{M} + \mathcal{M}_c$.

The marginal cost of unnecessary variables in research code is very common, as precise allocation and deallocation of data pointers in the Python code need a great command of the Python Language. A common problem is that few researchers notice to free the variables in each step and epoch. We take the code snippet below for an example.

```
for epoch in range(epochs):
  for step in range(_len_ // batch_size):
```

[14]The size of the layers are usually much smaller than the output data size of the layers.

```
    batch = ..
    out = model.forward()
    loss, logits = out[:2]
    loss.backward()
    optimizer.step()
```

The problem with this code is the variables allocated are still in the memory after each step ends. Note that these variables are in the CUDA memory instead of CPU memory because it would take too long for data to be transferred back and forth between GPU and CPU. Thus, the allocated CUDA memory cannot be freed. After each step, the GPU program not only recalculates the variables and replaces the original one, but also caches the original value in case of future use, leading to a serious memory expansion.

To solve this problem, an examination of what variables are safe to be deleted should be conducted, and then the researcher should delete the structures and free the GPU cache. After that, it's recommended to use `del` followed by the `empty_cache()` command. For example, the correct fix for the code above is adding these lines at the end of each step:

```
# epoch:
  # step:
    del out, loss, logits
    torch.cuda.empty_cache()
```

The `empty_cache()` command explicitly tells the CUDA compiler (`nvcc`) to clear the cache in memory. This command cannot be used without deleting the variable, because the run-time program is not able to find what cache to clear - both lines should be inserted at the same time.

As shown in Figure 17a, this approach has a drawback: the loss-time curve loses non-trivial performance for multi-GPU training. From the comparison of the experimental results, we see that clearing the memory in multi-GPU training takes lots of time. This is because flushing the cache will force the program to call a cleaner handler each time.

Conclusively, it's still recommended to use this strategy only for small models. For a more detailed explanation to multiple-GPU programming, you can refer to Section D.7 and Section D.8.

D.5 Change to CPU Memory

Changing the training process from GPU to CPU is another choice. As mentioned in C.1, the general memory is much larger than the GPU memory, so it's harder to run out of memory. However, training on the CPU is extremely slow, so we recommend only placing a small bunch of memory-consuming computations on the CPU.

D.6 Minimize Model

Minimizing the model is one of the most difficult strategies for reducing $\mathcal{M}$. It's hard to conduct because the model varies from one to another, leading to uncertainty [20]. In fact, this approach has even branched a subject itself, called model compression [5].

The basic strategies include changing two-level LSTM to one-level, replacing LSTM with GRU, reducing the number of convolution cores, and using as few `Linear` layers as possible [23].

D.7 Multi-GPU Data Parallel

Data parallelization focuses on using multiple GPU workers to distribute the data to reduce $\mathcal{M}$. If we represent the number of GPUs working together as k, the memory usage $\mathcal{M}_i$ for each GPU worker i now becomes the formula below.

$$\mathcal{M}_i = p_m \cdot n + \frac{b \cdot p_o}{k} + \frac{p_b}{k} \tag{26}$$

As we see, for output-dense training (large p_o), this multi-GPU approach decreases $\mathcal{M}$ by a factor of approximately k.[15] However, also note that $\frac{b}{k}$ must be an integer larger than one. If the batch size is 4 for 1 GPU, and 8 GPUs are available, then at most 4 GPUs can be used to apportion batch size, and the left 4 will get no data.

Here is another guess of the OOM error observed in Section 4.2. In the experiment on DPS with each GPU apportioned batch size of 4, we found that the training process ran into an OOM error, but the benchmark with 1 GPU apportioned batch size of 4 did not.

According to Formula 26, together with the condition that we have the batch size on each GPU the same as the benchmark (4), the deducted memory usage on each GPU should be as described in Formula 27, which is exactly the same as Formula 24, indicating the same memory usage with the benchmark.

$$\mathcal{M}_i = p_m \cdot n + \frac{b \cdot p_o}{4} \cdot 4 + \frac{p_b}{4} \cdot 4 \tag{27}$$

Except for the reason that the AllReduce algorithm still requires a master GPU to communicate, we suspect that p_o (sum of output data for each layer) is not averagely distributed, as described in Formula 28. We also take consideration of the marginal memory cost due to the engineering issue discussed in Appendix D.4.

$$\mathcal{M}_i' = p_m \cdot n + \frac{b \cdot (p_o + \Delta p)}{4} \cdot 4 + \frac{p_b}{4} \cdot 4 + \mathcal{M}_e \tag{28}$$

In the formula, Δp_i is the sum of the difference of the output data for each layer in the model on each node, and $\mathcal{M}_e$ is the memory usage caused by engineering work like no memory deallocation and cache deletion (which was illustrated in Appendix D.4).

D.8 Multi-GPU Model Parallel

The second type of multi-GPU strategy is the model parallel strategy. Different from the data-parallel strategy, it stores different parts of a model in different GPUs, with similar approaches like dispatching some data to the CPU for parallelization, as illustrated in Section D.5. It's pretty important for the researcher to figure out which part of the model causes the OOM issue and then apportion that part to another GPU for storage and computation.

[15] For those models with a relatively small p_o, the effect may be decreased but still worth a try.