597 Final Project: ADEM-VL

Siddharth Anmalsetty
ssa5526@psu.edu

Figure 1. An example of common VL tasks; captioning (top) and VQA (bottom)

1. Task

The primary goal of ADEM-VL [1] is to improve upon the efficiency of methods of solving VL (Vision/Language) problems such as VQA (Visual Question Answering), captioning, or instruction following. The two main difficulties in VL that ADEM-VL seeks to resolve are as below:

High Computational Cost

Integrating a visual aspect into an LLM leads to a much more complex input sequence than simply some text. Normally this is done via the use of another model to extract visual data (e.g. CLIP) and then fusing those layers together. This process requires far more advanced hardware than what each one would require in isolation. This also leads to the 2nd challenge.

High Memory Complexity

The most common way that models handle both inputs is through the use of multiple cross-attention modules. Each module brings with it significant parameters, leading to massive bloat when it comes to parameter count and thus model size. This heavily increases model training and inference time.

ADEM-VL is able to significantly reduce parameter count for higher-performance models, which in turn leads to much faster training and inference times. This results in a much more accessible and usable model that sacrifices little to no performance compared to any of the top methods.

2. Related Work

Flamingo

Flamingo is considered a seminal work for multimodal architectures and provides the basis for most multimodal architectures in vision and language. It uses gated cross-attention mechanisms for the integration of text into visual features. Attention scores are then computed to align the textual and visual features. As it is one of the first models to attempt to achieve this goal of integrating vision and language, it has some significant drawbacks. Since there is a cross-attention layer at almost every block, there is an extremely high inference cost to each step as well as an extraordinarily high number of trainable parameters. [2]

VL-Adapter/LLaMA-Adapter

VL-Adapter and LLaMA are strong examples of the goal of PEFT or parameter-efficient fine-tuning. Given that large models with high parameter counts take significant time to train and then tune afterward, there is a significant advantage in increasing training and inference speed. Each model uses vision adapters to transform visual input into a tokenized input that can be combined with textual tokens. Only these adapters are continuously trained leaving less workload on the LLM and focusing effort on simplifying visual input. While this does significantly reduce parameters as it is only processing “textual” input, there are significant issues present in computation. With the concatenation of the visual token and the text token, input sequences become incredibly long, which leads to quadratic

increases in the necessary computation. This also struggles with complex tasks such as VQA. [3]

LaVIN

LaVIN or LLaMA-Based Vision-Language Fine-Tuning is a model that extends the LLaMA LLM to vision language tasks. This is especially relevant to this paper as ADEM-VL also utilizes LLaMA. LaVIN uses a CLIP or similar encoder to extract visual features and then passes them through an adapter. These modules as mentioned before reduce trainable parameters. After passed through the adapter, features are then concatenated and evaluated at intermediate cross-attention layers. Since only the adapters are being fine-tuned, the model is a bit more efficient in terms of parameter usage. The problem LaVIN faces is that there is often visual feature overload as all features are loaded through the adapters regardless of importance or relevance which adds overhead. [4]

LoRA

Low-rank Adaptation or LoRA is a method originally designed for only language-based tasks. It can be used to move over an existing model to a new task without much overhead. Given its nature as simply moving another model over, it has an extremely low parameter count. LoRA works by splitting model weights into two parts, one large frozen part that serves as a baseline, and two smaller low-rank matrices. These two smaller matrices are then trained and then added back onto the original large matrix. This allows for extremely parameter-efficient training and can be applied to high-performing models in any field. LoRA does struggle with actual performance as it will never be as accurate as simply training a model from scratch. The model is also limited as there is no fusion of visual and textual features which leads its predictions to often be missing context. [5]

ADEM-VL

The reason why ADEM-VL can be considered the SOTA method for general Vision Language tasks is not due to its high performance. It performs relatively up to par with other models, it's about the fact that it is more accurate than other models with similar parameter counts for tasks like VQA, as well as being trained much faster and performing inference quicker. For tasks like captioning, ADEM-VL can display similar results to SOTA papers with a fraction of the parameters. The paper mentions a 15% training speed increase and a 3% inference speed increase. These improvements in training times make it far easier to train and tune models which makes this architecture SOTA in a different way than raw performance. Results for ADEM-VL in comparison to other models can be found in the results section. [1]

3. Approach

High-Level Overview

ADEM-VL, in essence, combines LLaMA for text information extraction and CLIP for visual information extraction and combines them using a variety of different innovative methods. They are as follows: parameter-free cross-attention, multiscale visual prompting, and adaptive fusion. These cross-attention layers are inserted into the LLaMA architecture and visual and textual layers are then aligned using a low-rank projection matrix.

Parameter-Free Cross Attention

With the interest of removing the large amount of cross-attention layers in other multimodal models and thus reducing parameter count, ADEM-VL attempts to create cross-attention layers in a nontraditional way.

$$\begin{aligned}\mathrm{XAttn}(\boldsymbol{X}_l, \boldsymbol{X}_v) &= \mathrm{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right)\boldsymbol{V} \\ &= \mathrm{softmax}\left(\frac{\boldsymbol{X}_l\boldsymbol{W}_Q\boldsymbol{W}_K^T\boldsymbol{X}_v^T}{\sqrt{d_k}}\right)\boldsymbol{X}_v\boldsymbol{W}_V\boldsymbol{W}_o^T,\end{aligned}$$

Original implementations for storing these relationships look like the following where X_t and X_v are textual and visual tokens respectively and the W's are projection matrices built off of features for determining relationships. As seen here, 4 projection matrices are generated for each set of tokens which is highly computationally intensive.

$$\mathrm{XAttn}(\boldsymbol{X}_l, \boldsymbol{X}_v)_i = \frac{\sum_j \mathrm{sim}(\boldsymbol{Q}_i, \boldsymbol{K}_j)\boldsymbol{V}_j}{\sum_j \mathrm{sim}(\boldsymbol{Q}_i, \boldsymbol{K}_j)},$$

Instead of this computationally intensive approach, ADEM-VL instead seeks to predict what these matrices would be without generation. This is done using the above function computes the similarity between a query vector Q, and a key vector K. The query vector is usually the textual token, and the key is the visual token. This is then taken one step further.

$$\mathrm{XAttn}(\boldsymbol{X}_l, \boldsymbol{X}_v) = \phi(\boldsymbol{X}_l)\phi(\boldsymbol{X}_v)^T\boldsymbol{X}_v,$$

As the similarity function resembles a dot product when expanded, the tokens can then be multiplied to find their cross attention. This is done by using the SiLU (sigmoid) activation function to line up the features and

allow for cross-attention. Effectively, visual features are now embedded into the textual space.

Multiscale Visual Prompts

Multiscale Visual Prompts have been shown to help other VL models in the past and were added here to try and circumvent the problems that may arise with reducing parameter count.

What effectively happens in this process is that a visual feature is scaled to different resolutions and then processed. This is done by applying pooling operations to images the get them to smaller resolutions and concatenating them. Features are then extracted using a CLIP encoder. Then comparing these features can get the relevant visual features. The best part is that this can be done in one single call of the vision encoder saving in computational costs.

Adaptive Fusion

While multiscale visual prompts may give all relevant visual features, many features will be irrelevant to the problem at hand. Adaptive Fusion seeks to resolve this problem by discarding any of this useless information.

This is accomplished first by generating the scores of the attention function mentioned before.

$$\text{XAttn}(\boldsymbol{X}_l, \boldsymbol{X}_v) = \phi(\boldsymbol{X}_l)\phi(\boldsymbol{X}_v)^T \boldsymbol{X}_v,$$

This allows the model to know which features are actually relevant based on how large the attention scores are. Attention scores are computed and stored in a matrix. A mask is then applied to the matrix where all scores below a certain amount will be dropped, thus removing a massive amount of irrelevant features. This can significantly improve performance without significant harm to accuracy.

Implementation

The primary code was using the ADEM-VL's original GitHub implementation. There were clear instructions on how to train and run the code, as well as attached model weights for their pre-trained versions. The dataset was downloaded from ScienceQA's official GitHub page and all files were organized as denoted in ADEM-VL's readme.

I created a Jupyter notebook to store and run all operations. Additional code was added to unzip files, copy larger files from my google drive into my colab environment, and create the proper directory hierarchy. I followed the instructions as given, adding in a few lines to adjust their code in very small ways to work properly. There were some errors in the readme that required some adjustments.

- Train.py requires an additional dummy argument so all proper arguments are passed to build.py properly.
- pycocoevalcap is not inside the requirements.txt file and is not built correctly so it requires a separate install
- The file hierarchy is incorrect in the readme and needs minor readjustments

Other than those things being resolved, the training and eval loop worked as intended.

4. Dataset

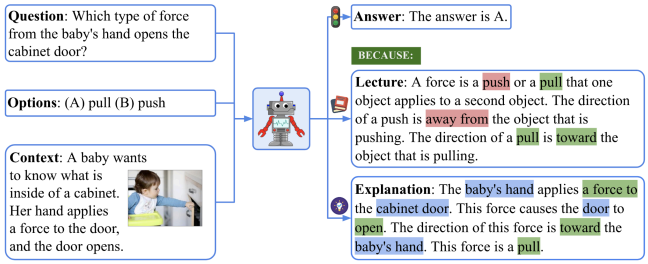

Figure 2: An example of a question from the ScienceQA dataset

The dataset used was the ScienceQA dataset. ScienceQA or SQA contains 21,028 different multiple choice questions based on different disciplines of science as follows: natural science, language science, and social science. The questions are sourced from the curriculum from the elementary to high school level. The test dataset contains about 2,178 test items and 18,850 training items. Data was downloaded directly from the ScienceQA github: https://github.com/lupantech/ScienceQA and no specific preprocessing was done on the dataset as it was small enough such that it was possible to use the entire thing.

The reason this was chosen as the test dataset is that it was the smallest of the three used in the paper and it had SOTA performance on this specific dataset.

5. Results

Table 1: Accuracy on ScienceQA dataset utilizing LLaMA-7B

Reported Accuracy from paper	Accuracy using given model weights	Replicated accuracy after training

93.85%	93.61%	85.81%

Given that ScienceQA is a multiple-choice dataset, there are no other real metrics to determine performance other than whether or not the model was able to predict the right answer for each test question. The difference between the paper's reported accuracy and the tested accuracy with the provided model weights is negligible. The small difference present is likely only a product of variance. A lower accuracy was expected considering that our trained model was trained on about ¼ of the epochs given the financial constraints of using Colab Pro.

For the above numbers, no hyperparameters were adjusted from the original settings other than reducing the number of GPUs being used (8 >> 1) and a corresponding increase in the gradient accumulation (4 >> 32) to achieve an equivalent effective batch size being used. This also allows for less memory usage and prevents running out of memory on a single GPU.

Hyperparameters

1. Max Sequence Length: 512 Characters
    a. The sequence length is the maximum number of characters a token can be before it's processed. 512 characters can be considered a generally agreed-upon number for a variety of NLP or Vision tasks considering it is big enough to take in most inputs while not being much too large. A smaller sequence length could miss out on more context.
2. Batch Size: 2
    a. This is how many samples can be processed in parallel. This is mostly irrelevant to the model as this is often a technical limitation, but increasing this number can likely increase training speed.
3. Accumulated Gradient: 16
    a. This means how many times a batch's gradient is accumulated per iteration. This is used to emulate larger batch sizes as the effective batch size in the current configuration is (2*16) = 32.
4. Epoch/Warmup Epochs: 5, 0
    a. This is simply how many training loops we will go through. Considering the difficulties in training, this cannot be changed reasonably in this current case.
5. Base Learning Rate: 9e-3
    a. The learning rate is how significantly the weights can change during training. Too low may cause no good training to occur. Too high of a learning rate can cause the training to be unstable and incorrect.
6. Weight Decay: 0.05
    a. Weight decay determines how big of a penalty large weight changes incur. This can help models generalize as weights will not be updated extremely. A larger one can be necessary if the model is tending to overfit, but considering the low number of epochs it may help the model out to lower this value.
7. Adapter Dimension: 12
    a. This determines how big the adapter layer is. The adapter layer modifies the pre-trained models for this task and a larger one would increase the parameter count, and likely the accuracy with cost to training efficiency.
8. Drop Ratio: 0.1
    a. This refers to how many nodes get "dropped' or masked in a neural network. This can prevent overfitting and issues relating to that. A 0.1 drop ratio means ten percent of all nodes are dropped. This is generally agreed upon to be a good amount and can be left alone.
9. Downsample Number: 256, 64
    a. This parameter determines how many images are downsampled before they are processed into the model from 256x256 to 64x64. This can be increased to add more complexity to the model or decreased to increase memory efficiency. As this is the maximum a song;e A100 GPU could handle, it was not changed from this value.

Reported Results in Paper

Table 3: Performance on ScienceQA dataset using LLaMA-7B model

Model	#Params	Accuracy
MM-Cot$_{large}$	733M	91.68%
LLaVA	7B	89.84%
LLaMA-Adapter	1.8M	85.19%
LLaMA-LoRA	4.4M	90.85%
LaVIN	3.8M	89.41%
ADEM-VL	4.5M	93.85%

MM-Cot and LLaVA are both fully trained models not using PEFT methods. The bottom 4 models are all PEFT and discussed in the related works section and ADEM-VL shows the best performance with a similar amount of parameters to other top performing models.

Table 4: Performance on MS COCO dataset for captioning

Model	#Params	BLEU-4	CIDEr
BLIP	583M	40.4	136.7
BLIP-2	188M	43.7	145.3
LaVIN	14M	36.2	122.2
LLaMA-Adapter	5.4M	37.8	131.7
ADEM-VL	5.5M	38.5	133.2

For captioning, BLEU-4 and CIDEr scores are used for calculating performance. BLEU-4 Measures precision of 4-grams in generated captions against reference captions and CIDEr considers consensus among multiple reference captions using tf-idf weighting. Generally higher scores are considered better here. BLIP and BLIP-2 are considered the SOTA in captioning and ADEM-VL can show comparable performance with much higher efficiency in parameter count.

Experiments

Experiment	Accuracy
Increasing Batch Size 2 → 4 Decreasing Accum-Iter 16 → 8	86.4%
Reducing Weight Decay 0.02 → 0.01	
Increasing Adapter Dim 12 → 16	88.4%

Rationale for Experiment 1: Reducing Weight Decay could increase overfitting, but considering the lesser amount of epochs this isn't as big of a concern. Increasing batch size can improve training speed stability as we aren't trying to emulate a larger batch size now. Reducing accum-iter here is to account for this larger batch size.

Rationale for Experiment 2: Increasing adapter dim would increase the parameter count but it would improve the performance of the model as the capacity would increase and it would be able to generate more relationships.

6. Possible Improvements and Results

Given the ease of training this model, identifying significant hyperparameter improvements was unlikely as the paper's authors have likely done many more experiments. While there were slight improvements to accuracy after the previous experiments, it was only on 5 epochs and can't easily be proven to be significant when extended to more epochs. As for other possible improvements to this technology some suggestions I have would be as follows:

Using a more advanced LLM for textual features

Instead of using LLaMA, perhaps integration with a GPT-4 like model or another could hypothetically improve performance. While this may increase the reliance on pretrained models, it would definitely have a positive impact on performance. A similar improvement can be made by using a more advanced pretrained model for visual feature extraction.

Improving regularization

The current method for regularizing and generalizing the model are very simple implementations of weight decay and dropout layers. A more advanced method could use slightly more technical techniques such as layer dropout or dropping out in the cross-attention layers as well.

Learnable threshold for Adaptive Fusion

Currently, when features are fused and their attention scores are below a certain point, they are discarded. Instead of using a fixed value, a dynamic threshold

could be introduced, which will drop out more or less tokens based on performance.

7. Code Repository

https://github.com/ssa5526/CSE597-Final-Project

The jupyter notebook is relatively self-explanatory and a short instruction set is there. I'd recommend following along with the ADEM-VL instructions simultaneously to make sure there isn't any confusion.

References

1) Hao, Z., Guo, J., Shen, L., Luo, Y., Hu, H., & Wen, Y. (2024). ADEM-VL: Adaptive and Embedded Fusion for Efficient Vision-Language Tuning. arXiv preprint arXiv:2410.17779.
2) J.-B. Alayrac, J. Donahue, P. Luc, A. Miech, I. Barr, Y. Hasson, K. Lenc, A. Mensch, K. Millican, M. Reynolds et al., "Flamingo: a visual language model for few-shot learning," Advances in neural information processing systems, vol. 35, pp. 23 716–23 736, 2022.
3) Sung, Y.-L., Cho, J., & Bansal, M. (2022). VL-Adapter: Parameter-Efficient Transfer Learning for Vision-and-Language Tasks. arXiv preprint arXiv:2112.06825.
4) G. Luo, Y. Zhou, T. Ren, S. Chen, X. Sun, and R. Ji, "Cheap and quick: Efficient vision-language instruction tuning for large language models," Advances in Neural Information Processing Systems, vol. 36, 2024.
5) E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," arXiv preprint arXiv:2106.09685, 2021.