UNIFIED-IO for Robotics Action Generation

Gregory LeMasurier
University of Massachusetts Lowell
Gregory_LeMasurier@student.uml.edu

1 Introduction

Multimodal NLP is a popular approach to solve vision-language tasks. These models learn a mapping of language to some other input modality, such as images. This is similar to when people learn a new language, or when children are learning to speak, as they learn representations between the language and the real world. The learned representations between language and visual inputs are derived from the same underlying reasoning for many vision-language tasks (Cho et al., 2021).

The robotics domain consists of several different vision-language tasks. By leveraging the similar vision-language reasoning skills across a variety of different robotics tasks, a singular model can be used as a generalized solution to a variety of tasks (Shao et al., 2021; Jiang et al., 2022). These models are called unified models.

Unified models enable a single architecture to solve numerous different tasks, in some cases even using a single set of parameters. Thus, these models are more generalizable across a variety of tasks. One popular state of the art unified model is UNIFIED-IO (Lu et al., 2022). UNIFIED-IO is a Seq2Seq transformer based model that can take in a variety of inputs and can generate a variety of outputs, such as text, images, masks, keypoints, and boxes. UNIFIED-IO has been applied to the following tasks: Image Synthesis from Text, Image Inpainting, Image Synthesis from Segmentation, Object Detection, Object Localization, Keypoint Estimation, Referring Expression, Depth Estimation, Surface Normal Estimation, Object Segmentation, Image Classification, Object Categorization, Webly Supervised Captioning, Supervised Captioning, Region Captioning, Visual Question Answering, Relationship Detection, Grounded VQA, Text Classification, Question Answering, Text Summarization, Masked Language Modelling, and many more.

2 Research Question

UNIFIED-IO (Lu et al., 2022) has been proven to generalize across a variety of input and output modalities in order to solve a variety of different vision-language tasks. Many tasks in the robotics domain can be represented as vision-language tasks, where sensor input comprises the vision element, and language can be used to represent a series of actions or commands. Thus, we hypothesize that UNIFIED-IO should be capable of generalizing to the robotics domain. Through this work, we finetune UNIFIED-IO on a simple pick and place style task and evaluate its performance.

3 Data

For this project, we compared the datasets used in two recent unified multimodal models in the robotics domain. The dataset used by VIMA (Jiang et al., 2022) was better fit for this project, as it was robot agnostic, could be linked to VIMA-Bench to benchmark the results, and was structured better for our goals. VIMA was a unified multimodal model for the robotics domain that focused on four levels of task generalizability for zero-shot generalization. These levels include Object Placement, Novel Combination, Novel Object, and Novel Task.

The VIMA dataset contains 650K trajectories for 13 different robotic manipulation tasks. Each trajectory contains an overhead and front scene image at each action step. Observations including each object mask and the robot's end-effector are included for each action step as well. The solution to each task is comprised of several action steps, the expected position and rotation of the robot's end-effector is provided for every step. Additionally they provide meta information for each task, including information such as object properties and prompts.

For this project we have focused on the Simple Object Manipulation category of tasks, specifically

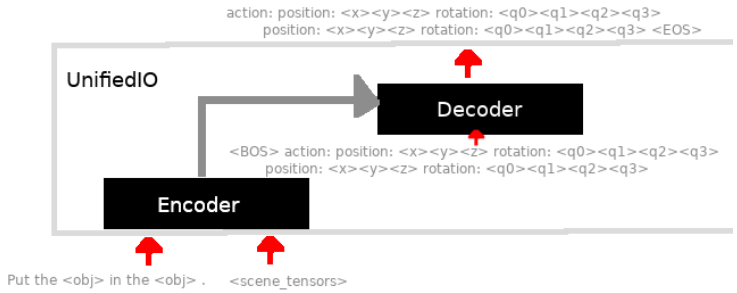

Figure 1: A high level overview of the UNIFIED-IO model for the Simple Manipulation Task.

Task-01, which is a simple pick and place task. This particular task contains 50062 trajectories. This data was then split into train, validation, and test sets. About 80% of the data was used for training, and the remaining 20% was used split evenly with 10% used for validation and 10% used for test. These are rough estimates as some garbage data was found after the data split, as will be discussed in the next section. Each expected action sequence for this task consists of two actions, one for pick and one for place.

3.1 Data Processing

The subset of the VIMA dataset that we are using for this project is roughly 140GB. It contains a lot of meta data that is not necessary for our particular application. Therefore we need to preprocess the data to extract the bare minimum information necessary for our task. This task contains four scene images, two from each view point. We only need the initial scene image as our model does not update its plan based on its previous actions, rather it predicts the full action sequence from the beginning. From the trajectory metadata we then need to extract the prompt. The prompt contains an object name which indicates which objects should be used. Using UNIFIED-IO's region tokenizer, we can generate object tokens to insert into the prompt. Each object token is a sequence of four special tokens that represent the bounding box of the object relative to the scene image. The prompt template for this task is "*Put the <obj> into the <obj>*."From the trajectory we also extract the action bounds for the range of the possible positions for the action space. For this particular task, the action bounds are from -0.5 to 0.75. Using this bounds, we then create a quantizer for the action poses. The quantizer has 100 slots evenly spread across the action bounds. Then we generate our expected action sequence. This sequence is in the format: "*<bos>action: pose: <x><y><z> rotation: <q0><q1><q2><q3> action: pose: <x><y><z> rotation: <q0><q1><q2><q3><eos>*" where the special tokens are generated by passing the expected values, given to us in the expected action sequence definition, to the quantizer. While processing the actions, we found 41 of the 50062 samples had two poses for each action. This doesn't make sense for the task, as each action is to one position. Therefore, these 41 samples were excluded from the cleaned data. All of this data is stored in a csv for each data sample. This enables us to quickly load the dataset every run.

4 UnifiedIO

UNIFIED-IO is a Seq2Seq transformer based model. The inputs are tokenized using a t5-base tokenizer that is available through the transformers T5Tokenizer. The tokenizer has a vocab size of 33200 where 1200 tokens are extra special tokens. Out of these 1200 extra tokens, 1100 are used by UNIFIED-IO's tokenization methods and 100 are used for our pose quantization. UNIFIED-IO's decoder length is set to 27, which is the maximum length of the action sequences for this task. Additionally, we set an image decoder length to 1, which is the minimum allowed as we do not generate images in this task. UNIFIED-IO has a small, base, large, and xl pretrained model. For this project, we focus on using the small and base models as they fit in my testing GPU. The small model is roughly 14 million parameters, while the base model is roughly 31 million parameters. UNIFIED-IO uses Jax (Bradbury et al., 2018) which greatly improves the performance due to the just-in-time compilation.

Figure 1 shows a high level overview of the inputs to UNIFIED-IO for the Simple Manipulation Task. The encoder takes in image tensors as well as a tokenized prompt. For training, we feed in

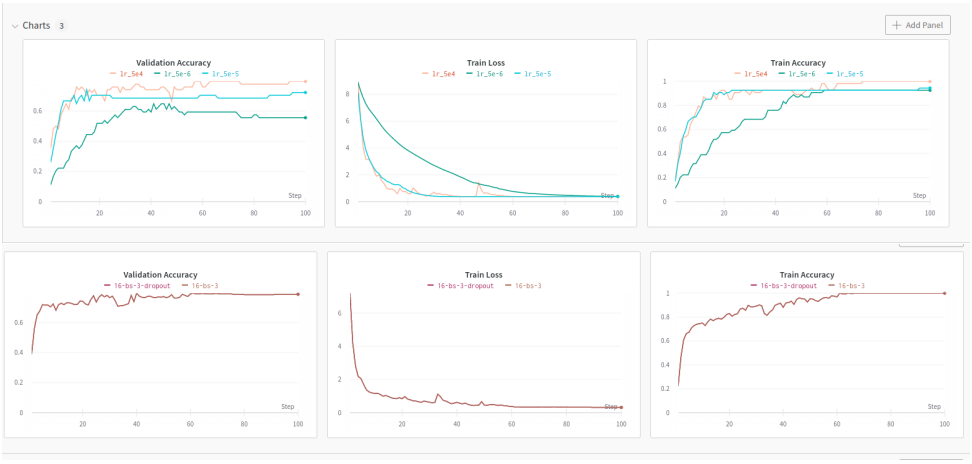

Figure 2: Graphs used to determine parameters using a small sample of 16 instances, small model, and batch size of 3.

a tokenized action sequence to the decoder, with a BOS token and not an EOS token. The model should then predict the next token given the previous tokens, so the output from the decoder is a set of logits representing which token should be next in the sequence. We select the token with the highest probability, resulting in a series of tokens similar to that shown in Figure 1. This output sequence does not have a BOS token, and ends with an EOS token. The decoder does also take in an image input, and produces an image output, however, for this task they are not necessary. Thus we use an empty image structure which then triggers a flag in UNIFIED-IO indicating that we are not generating images. We ignore all outputs from the decoder, except for the text logits.

4.1 Training Loop

UNIFIED-IO has not publicly released their fine-tuning or training software, therefore, we had to develop a training loop from scratch. The training loop is a Jax based training loop, which uses just-in-time compilation for the training and validation steps as well as the computation of our metrics. This drastically saves time, as the validation loop would take roughly an hour, however, with this compilation it would take less than 10 minutes. We also created a custom torch dataloader. This drastically reduced memory consumption, as we only needed to load the images that are being used in the current batch.

4.1.1 Parameter Selection

There are three parameters that we need to determine before running our training loop. To find the parameters, we ran small tests using a small UNIFIED-IO model configuration, used 16 samples of data for training and 16 for evaluation, and ran each test for 100 epochs. The first parameter we identified is batch size. The largest batch size that I could fit on my testing machine is 3, therefore all further parameter analyses were conducted with a batch size of 3. When using the google cloud machines, I used a batch size of 32 with the small model configuration and 16 with the base model configuration.

Next, we identified the optimal learning rate. To determine the optimal learning rate, we looked at the shape of the graphs and if the model was able to successfully over-fit to a small sample of data. To do so I compared learning rates of $5e-4$, $5e-5$, and $5e-6$. In the top graph in Figure 2 we can see that a learning rate of $5e-4$ properly over-fit at 74 epochs, therefore, we will use this learning rate for all further runs.

Finally, we need to identify dropout. UNIFIED-IO has dropout built in, therefore we just need to enable it by passing in a flag. The bottom graphs in Figure 2 show that enabling dropout did not produce differing results with a small sample. This was unexpected, future work should ensure that UNIFIED-IO is handling the dropout as we expect.

We then used these parameters to train a small and base UNIFIED-IO model on the full training set of data. Each model was trained for a total of 50 epochs.

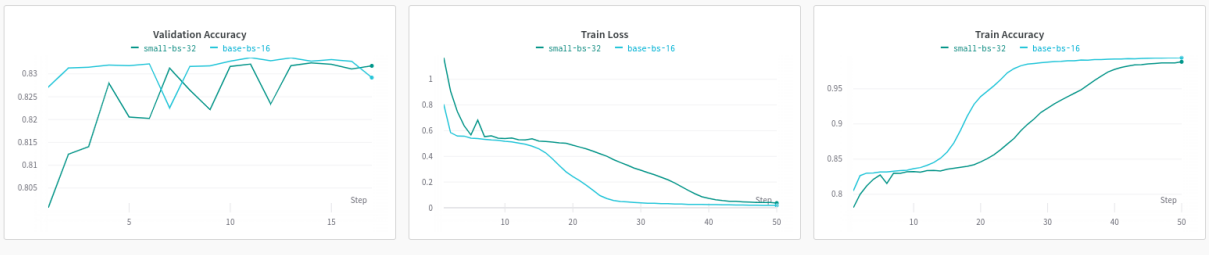

Figure 3: Validation Accuracy, Training Loss, and Training Accuracy for the fully trained models.

Model	Accuracy	Position Token Accuracy	Euclidean Distance (3D)	Euclidean Distance (2D)
small	83.29%	43.38%	**0.04682**	**0.04609**
base	**83.34%**	**43.56%**	0.04845	0.04764

Table 1: The action sequence accuracy, position token accuracy, and euclidean distance achieved by our optimal small and base models on the test set.

5 Results

5.1 Training

After fully training the small and base models for 50 epochs, it is apparent that the models began to over-fit to the training data. Thus, we selected the optimal checkpoint for the model before they started to over-fit. For the small model, this was at epoch 14 with a validation accuracy of 83.24% as can be sen on Figure 3. Accuracy is the average sequence matching percent, compared to the expected action sequence. For the base model, the optimal checkpoint was at epoch 11, with a validation accuracy of 83.36%. These models were used for all further analyses.

5.2 Evaluation

After selecting the optimal checkpoints, we the evaluated the test set for sequence token accuracy, the accuracy of the position tokens, euclidean distance of the position tokens, and the impact of quantization on the euclidean distance.

5.2.1 Accuracy

The first metric we analyze is the sequence match accuracy. This metric is the same as the accuracy used for training and validation, as it represents the average percentage of matched tokens compared to our expected decoder outputs. The small model achieved an accuracy of 83.29% whereas the base model achieved an accuracy of 83.34%. These results can be foud in Table 1. This was surprisingly low, especially considering how 13 of the 27 tokens are constant for this task. Out of the remaining 14 tokens, six are for position values, and 8 are for rotation values. The rotation values are not important for this particular task, therefore we really only care for the six position tokens that the model is generating.

5.2.2 Position Token Accuracy

As the test accuracy was lower than expected, we evaluated the accuracy of only the position tokens, since these are the most important tokens. To our surprise the position token accuracy was also very low. The small model achieved 43.38% position token accuracy and the base model achieved 43.56%, as seen in Table 1. Despite the fact that these accuracies are low, this could be due to the quantization. Each position element is quantized into one region, even if the position is really close to the expected quantized region, if it does not fall directly in its bounds it will be in a neighboring region. Several neighboring regions can all produce valid solutions to this problem, as the accuracy for picking and placing is not needed to be extremely precise. To test if this was the case, we evaluated the Euclidean distance from our predicted and expected positions.

5.2.3 Euclidean Distance

The Euclidean distance will tell us the distance from our predicted positions to the expected positions. Unlike the token match comparison, this metric shows us how close our model really was to the expected action sequence. First we evaluated the Euclidean Distance for the 3D position that the model generated. The Euclidian distance ranges from 0 meters, which is an exact match, to 2.165064 meters, which is the maximum distance in the action bounds. Our small model had an

average distance of 0.04682 meters, and the base model had an average distance of 0.04845 meters, as shown in Table 1. These results look much better. As the distances from predicted to expected poses are low, it appears that the accuracy was low due to neighboring quantized regions being selected, which could perfectly solve the task as well.

The z position is pretty hard to predict from a single top view of a scene, as this is the depth component of the position. Thus we also evaluated using a 2D Euclidean distance metric, where we only consider the x and y positions. The range for this metric is 0 to 1.767767 meters, as determined by the action bounds. Our small model had an average 2D distance of 0.04609 meters, and the base model had an average 2D distance of 0.04764 meters, as shown in Table 1. In this particular task, the expected heights are all very similar, thus, the distance did not change that much.

5.2.4 Impact of Quantization

Our predicted position tokens are all decoded to generate their corresponding value. Due to the quantization, it is not possible to get zero Euclidean distance. This is because we lose precision as ranges are used to determine the appropriate quantization region. Thus, we have a constant quantization error that will influence our results. To identify this quantization error, we compute the Euclidean distance from the quantized expected positions to the raw expected positions. The average quantization error is 0.01104 meters, thus if our model had perfectly generated the same quantized regions as the decoder expected, we would still have a Euclidean distance of 0.01104 meters, not zero meters, due to the loss of precision.

5.2.5 Comparison to SOTA

VIMA (Jiang et al., 2022) results were analyzed on VIMA-Bench. The state of the art method for Task-01 with L1 generalization is VIMA which completed the VIMA benchmark with 100% accuracy, as seen in Table 2. The results reported in this table were of the 20M models, as they were closest in size to our small and base models. Unfortunately, due to time restrictions we were not able to evaluate our model on the VIMA benchamarks. Therefore, we report our average error through Euclidean distance. This error is very low, it is likely that the robot would successfully complete the simple manipulation task with a high success rate. We also only show L1 generalization as we would need to interface with VIMA-Bench for the other levels of generalization, whereas L1 can be done with a sample of the VIMA dataset. Future work should evaluate this model on the VIMA benchmarks.

Method	Task 01 - L1 Generalization
VIMA (20M)	**100%**
Gato (20M)	62%
Flamingo (20M)	56%
Decision Transformer (20M)	59.5%
*UNIFIED-IO (small)(14M)	*Average Error: 0.04682 m*
*UNIFIED-IO (base)(31M)	*Average Error: 0.04845 m*

Table 2: State of the art results on the VIMA benchmarks. NOTE: UNIFIED-IO was not yet run on the VIMA benchmarks, thus we report our average error for each position in our test set.

6 Conclusion

Overall, the UNIFIED-IO results look promising on VIMA Task-01. Though we couldn't evaluate on the VIMA-Bench, our average position error is very low, implying that the robot would have a high success rate on this task.

Throughout this project I learned a lot. In particular, I learned about the state of the art and cutting edge work in multimodal NLP. I also learned about Jax, and how it can be used to drastically reduce run time due to its just-in-time compilation. I was able to implement my training and validation loops using Jax's just-in-time compilation. My evaluation loop was not able to leverage jax, due to it's dependencies on my input data, therefore, it ran much slower than the validation loop which has a comparable number of steps. I also learned how to efficiently load large datasets using torch dataloaders. Initially, I would exhaust my memory trying to load too many images at once, however, once I created a custom dataloader, I was able to only load the images that the current batch needs.

Future work could investigate integrating the model with VIMA-Bench to get results that can be compared to the state of the art methods and across the four generalization levels. Additionally, future work could investigate predicting a single action at a time, taking in changes after each action step to predict the next action. This method can then be compared to the approach used in this project,

where the entire action sequence is generated at the start. While the approach used in this paper works in static environments, integrating observations at each action step is necessary for dynamic environments as the robot may need to adjust its plan due to changes in its environment.

References

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: composable transformations of Python+NumPy programs.

Jaemin Cho, Jie Lei, Hao Tan, and Mohit Bansal. 2021. Unifying vision-and-language tasks via text generation. In *International Conference on Machine Learning*, pages 1931–1942. PMLR.

Yunfan Jiang, Agrim Gupta, Zichen Zhang, Guanzhi Wang, Yongqiang Dou, Yanjun Chen, Li Fei-Fei, Anima Anandkumar, Yuke Zhu, and Linxi Fan. 2022. Vima: General robot manipulation with multimodal prompts. *arXiv preprint arXiv:2210.03094*.

Jiasen Lu, Christopher Clark, Rowan Zellers, Roozbeh Mottaghi, and Aniruddha Kembhavi. 2022. Unified-io: A unified model for vision, language, and multi-modal tasks. *arXiv preprint arXiv:2206.08916*.

Lin Shao, Toki Migimatsu, Qiang Zhang, Karen Yang, and Jeannette Bohg. 2021. Concept2robot: Learning manipulation concepts from instructions and human demonstrations. *The International Journal of Robotics Research*, 40(12-14):1419–1434.