Improving the Efficiency of 3D Pose Estimation Model

Quan Ho
Stanford University
quanmho@stanford.edu

Abstract

The increasing interest in virtual reality and digital avatars has necessitate faster and cheaper methods for converting physical motions into the digital space. While high-fidelity motion capture systems offer precise 3D motion reproduction, they are expensive and require substantial setup. Alternatively, RGB videos can easily be captured with minimal setup and then converted to motions in the digital world using 2D keypoint detection techniques and 2D-to-3D lifting approaches. This paper explores 2D-to-3D lifting approaches for inferring 3D motion from 2D videos, focusing on optimizing existing methods. Specifically, it investigates the impact of training a 2D-to-3D lifting model, namely the HuMoR model, using the FP16 format. Evaluation on training the HuMoR model with FP16 format reveals that while FP16 training offers modest speed improvements, its benefits are more pronounced with larger models. Additionally, FP16 training does not significantly affect loss values compared to FP32 training. Further work to follow up on this paper could be exploring FP16 training with other larger models and investigating even smaller formats such as FP8.

1. Introduction

Having the ability to interact with the digital world using motions has long been a desired goal for researchers and consumers alike. Humans have always interacted with the physical world through our motions rather than through a keyboard and a mouse. Human motions can also be a form of visual communication that can convey meanings and feelings difficult to convey with texts. With the rise of interests in virtual reality and digital avatar, we will need methods to translate motions from the physical world to the digital world that are not just accurate but also fast and accessible. High-fidelity motion capture system can reproduce highly accurate 3D motion, but such system is expensive and require large empty area to setup. On the other hand, RGB videos are readily available and can be captured by anyone with minimal setup. There are two categories of approaches for inferring 3D motion from a 2D video: (1) direct estimation, and (2) 2D-to-3D lifting. direct estimation approaches attempt to infer 3D motion directly from 2D images or videos, whereas 2D-to-3D lifting approaches leverage existing 2D pose detectors to estimate the 2D pose then use a different model to infer 3D pose from the predicted 2D pose. In general, 2D-to-3D lifting approaches tend to outperform direct estimation approaches [8]. Thus, in this project, we will focus on optimizing existing 2D-to-3D lifting approaches. In particular, we will explore the impact of training 2D-to-3D lifting models using FP16 format.

1.1. Related Work

To find a 2D-to-3D lifting approach for optimization, we explored earlier works of different 2D-to-3D lifting approaches and summarized our findings below.

PoseFormer. PoseFormer is an early attempt at leveraging transformer architectures for 3D human pose estimation by using a purely transformer-based approach with no convolution architectures involved [8]. PoseFormer is consisted of three transformer modules: spatial transformer module, temporal transformer module, and regression head module [8]. The spatial transformer module is for extracting the local joint relations within each frame. The temporal transformer module is for extracting dependencies across a sequence of frames. Finally, regression head module is for predicting the 3D pose of the center frame using a sequence of frames.

PoseFormerV2. PoseFormerV2 is an improved version of PoseFormer. Unlike PoseFormer which used a temporal transformer module, PoseFormerV2 represents the sequences in frequency domain, where high-frequency components can be dropped to scale up the receptive field and boost robustness for 2D joint detection with noisy data. PoseFormerV2 converts the sequences into Discrete Cosine Transform (DCT) coefficients and replaces the temporal transformer module with a time-frequency transformer that extracts features from both the time domain and the frequency domain [7]. Compared to PoseFormer, PoseFormerV2 has both better speed and accuracy.

HuMoR. Instead of the typical approach of estimating 3D motion from videos, HuMoR is a learned, autoregressive, generative model that captures how pose changes over time and models a probability distribution of possible pose transitions using conditional variational autoencoder [4]. HuMoR is trained on the AMASS motion capture dataset [1]. At test time, HuMoR is used as a motion prior along with a robust test-time optimization strategy to enable 3D human motion estimation from noisy and partial observations across different input modalities such as RGB/RGB-D video and 2D or 3D joint sequences [4].

1.2. Mixed Precision Training

As ML models become larger and more complex, the computational resources required for training the models and performing inference also grow. To address this issue, mixed precision training techniques have been developed to enable training with lower-precision data formats like FP16. Compared to FP32 which uses 32 bits, FP16 only uses 16 bits, so models using FP16 format will require less memory. Furthermore, switching to FP16 also shortens training and inference time by reducing memory access time as well as being able to leverage hardware dedicated to accelerating FP16 arithmetic throughput such as NVIDIA Tensor Cores [2]. Mixed precision training techniques were used to retrain several CNNs (AlexNet, VGG-D, GoogLeNet, Inception v2, Inception v3, pre-activation Resnet-50) for ILSVRC classification task and achieved similar accuracy compared to the baseline FP32 training sessions [2]. Similarly, detection CNNs (Faster-RCNN, Multibox-SSD) were retrained with mixed precision techniques and also achieved similar accuracy as the baseline FP32 training session.

2. Methods

HuMoR Details. We evaluate the effect of training on FP16 format by re-training the HuMoR model with Pytorch Automatic Mixed Precision (AMP) recipe. Due to time constraint, we could only pick one model to re-train, and we choose HuMoR because the model code is publicly available and well-documented. HuMoR takes in and outputs matrix representing the state of a moving person. Specifically, the state matrix $\mathbf{x}$ is composed of a root translation $\mathbf{r} \in \mathbb{R}^3$, root orientation $\Phi \in \mathbb{R}^3$ in axis-angle form, body pose joint angles $\Theta \in \mathbb{R}^{3\times 21}$ and joint positions $\mathbf{J} \in \mathbb{R}^{3\times 22}$:

$$\mathbf{x} = [\ \mathbf{r}\ \ \dot{\mathbf{r}}\ \ \Phi\ \ \dot{\Phi}\ \ \Theta\ \ \mathbf{J}\ \ \dot{\mathbf{J}}\ ], \tag{1}$$

where $\dot{\mathbf{r}}, \dot{\Phi}$ and $\dot{\mathbf{J}}$ denote the root and joint velocities, respectively, giving $\mathbf{x} \in \mathbb{R}^{3\times 69}$ [4].

For model architecture, HuMoR uses a conditional variational autoencoder (CVAE) [5] for predicting the next pose state $\mathbf{x}_t$ given the previous pose state $\mathbf{x}_(t-1)$. The latent space variables $\mathbf{z}_t$ can then be viewed as the probability of transitioning to the next pose state given the previous pose state, which is denoted with $p_\theta(\mathbf{x}_t|\mathbf{x}_{t-1})$. Since different poses will have different distributions of possible next pose (e.g, an idle person would have many possible next states, but a person in midair would have very little possible next states), rather than having a prior with solely a Gaussian distribution, HuMoR performs training on the prior to yield a conditional prior to output different distributions based on the given pose state [4]. The decoder part of the architecture outputs two outputs: the change in pose state Δ_θ and the person-ground contact probability $\mathbf{c}_t$ for each of the eight body joints. The encoder, conditional prior, and decoder are all 5-6 layer MLPs with ReLU activations and group normalization [4, 6], which is the part of HuMoR that we are interested in re-training in FP16. HuMoR's CVAE was trained using the typical approach for training CVAE which is to maximize the likelihood lower bound:

$$\log p_\theta(\mathbf{x}_t|\mathbf{x}_{t-1}) \geq \mathbb{E}_{q_\phi}[\log p_\theta(\mathbf{x}_t|\mathbf{z}_t, \mathbf{x}_{t-1})] \\ - D_{\text{KL}}(q_\phi(\mathbf{z}_t|\mathbf{x}_t, \mathbf{x}_{t-1}) \parallel p_\theta(\mathbf{z}_t|\mathbf{x}_{t-1})). \tag{2}$$

Profiling Traces. To understand the impact of training with FP16, we would like to measure the training time using Python timer, but this approach quickly became inadequate due to not having enough granularity. Instead, we utilizes PyTorch profiler tool to profile a single step of propagating one batch of size 8 of data through the model and computing the loss. The profiler tool outputs a trace file with the execution order and timing of functions and GPU kernels that were used. We set the profiler tool to map each portion of the execution trace to a specific part of the training algorithm, namely, preparing the data, sampling the data (propagating the data through the MLPs to obtain next pose state), and computing the loss.

3. Dataset

We re-train the models using the AMASS dataset [1] used by the original authors. However, due to our system constraint, we cannot use the entire AMASS dataset, so we opted to only train with the CMU dataset. For validation, we used MPI_HDM05, SFU, and MPI_mosh. For preprocessing the dataset and extracting features, we used the provided pre-processing script in the official HuMoR codebase.

4. Experimental Results

We evaluate the impact of training with FP16 based on the execution time and the loss value after training for 8 epochs. Our system uses NVIDIA T4 with 16GB memory, and due to time constraint, we chose to observe execution time over only 2 epochs and observe the loss over only 8 epochs instead of 200 epochs that were used to train the

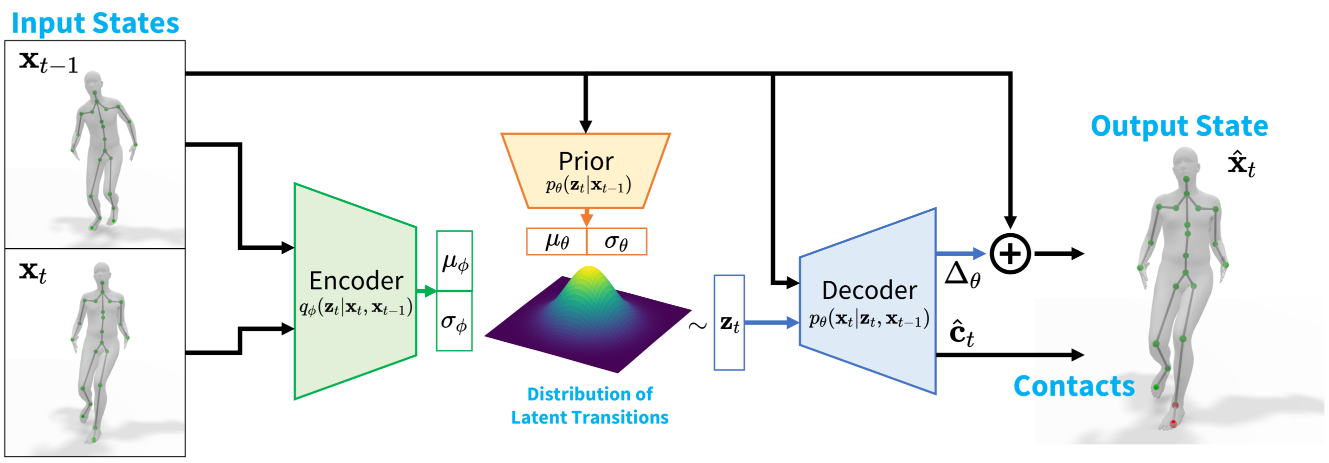

Figure 1: *HuMoR CVAE Architecture* [4].

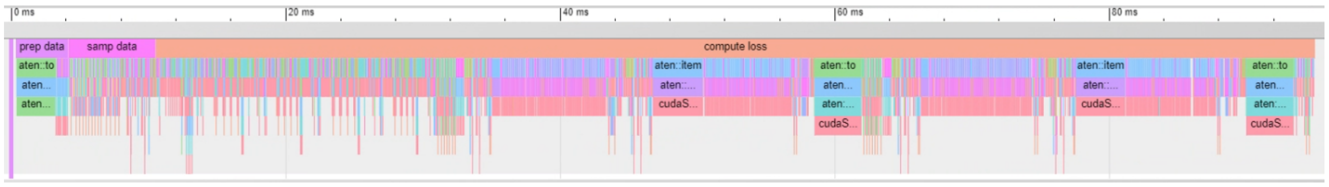

Figure 2: *Execution Trace.* PyTorch profiler tool outputs a trace file that can be viewed in Google Chrome. This figure shows the execution of taking one single training loss step using FP16 data format. In particular, the majority of the execution time is spent in evaluating the loss function.

.

	Execution Time over 2 Epochs
FP32 Training	8h 52m (532m)
FP16 Training	8h 32m (512m)

Table 1: FP16 finishes 2 epochs of training 20 minutes faster than FP32, which equates to 4% improvement in execution time.

	FP32	FP16
Prepping Data	3.80ms	4.11ms
Sampling Data	6.36ms	8.42ms
Computing Loss	84.47ms	98.05ms

Table 2: Across the three steps in one single training loss step, FP16 shows slight slowdown.

original HuMoR model [4]. We keep all the other training hyperparameters identical with the original HuMoR training algorithm.

Total Execution Time. We simply wrapped the original training code for HuMoR with PyTorch AMP recipe without other modification for our first investigation, and we saw only an improvement of 4% in execution time. We expected significantly more speedup since training with smaller data format and being able to use dedicated accelerated hardware for FP16 should at least improve computation time.

To further understand the lack of speedup, we use PyTorch profiler tool to dump out execution trace for the first training loss step. We tried profiling beyond the first training loss step, but the system would crash every time we profile beyond the first training loss step. We record the execution time for each part of the training loss step in Table 2 and a visualization of the execution trace in Figure 2.

From the execution trace breakdown for the first training loss step, we see that the majority of the computation time is spent in evaluating the loss function. This would partly explains the lack of speedup from training with FP16 because the loss function doesn't benefit as much from the speedup for matrix-multiply-add operations when training in FP16. The part that would benefit the most from hardware-accelerated matrix-multiply-add in FP16 would be the sampling data step with all the MLP layers, but that part only represents 4-5% of the total execution time of one training loss step, hence the speedup becomes less noticeable. However, looking at the execution time breakdown for the first training loss step, we see that there are slowdown across all the steps. To understand the slowdown better, we profiled the sampling step and the loss function in more granularity.

Sampling Step Execution Time. We zoom in and take a closer look to how the sampling step is being done. In particular, we look at how the CPU and GPU are spending

	$\text{FP32}_{\text{4xMLP}}$	$\text{FP16}_{\text{4xMLP}}$
Sampling Data	17.11ms	9.00ms

Table 3: For sampling step in one single training loss step with 4x MLP dimensions, FP16 now shows an almost 2x speedup.

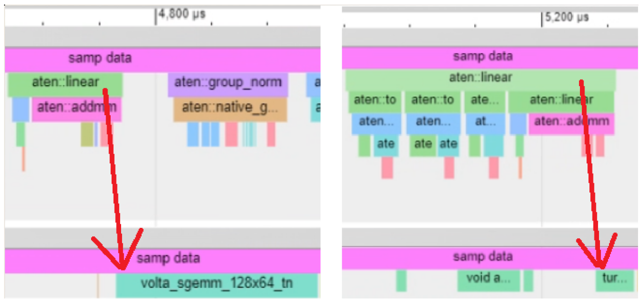

Figure 3: *1024x1024 Linear Layer in MLP.* This shows the execution traces (Left: FP32, Right: FP16) for a forward pass through one linear layer of input size of 1024 and output size of 1024. The red arrow indicates the corresponding GPU kernel for the linear layer, where the computation are being done. Everything before is overhead logic to copy the data into GPU and, for FP16, also logic to convert FP32 data to FP16.

their time when doing one forward pass through a single 1024x1024 linear layer. If we only look at the GPU kernel at the bottom of Figure 3, we see that FP16 are almost 8x faster than FP32. This matches our expectation that FP16 would provide significant computational speedup. However, this speedup is not observed at levels above the GPU kernel because FP16 has an extra fixed cost to convert FP32 data to FP16 format which is more apparent when the dimensions of the linear layer is too small to bring GPU utilization to 100% so most of the execution time is spent in overhead logic instead of computation. We tried re-training the model by quadruple all the MLP layers, bringing the parameter count from 9.7 millions to 133 millions, to see if increasing the computation requirement would make the speedup benefit more visible. The result is that after quadrupling the MLP layers, we see almost 2x speedup for the sampling step. This indicates that training at FP16 is more beneficial when training large models as compared to training small models.

Loss Function Execution Time. We did a similar investigation into the slowdown observed with the loss function as we did for the MLP layers. As seen with the MLP layers in the sampling step, the GPU has very low utilization during the loss function evaluation step, resulting in paying extra cost for overhead logic to use FP16 without being able to benefit from the FP16 accelerated hardware. Improving GPU utilization for the loss function is more complex than for the MLP layers since we cannot simply increase

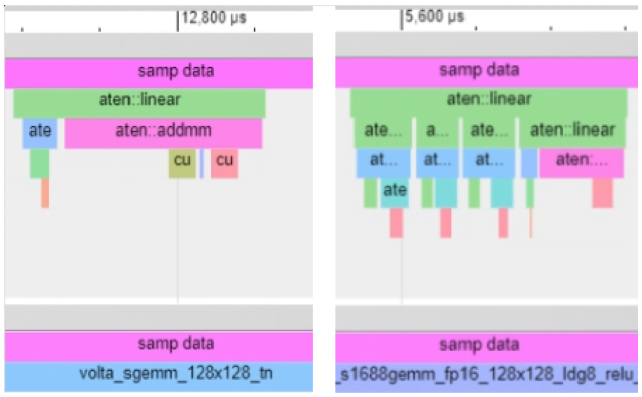

Figure 4: *4096x4096 Linear Layer in MLP.* This shows the execution traces (Left: FP32, Right: FP16) for a forward pass through one linear layer of input size of 4096 and output size of 4096. With larger linear layer size, we now see the GPU spending more time doing the computation according to the bottom bar, hence increasing the GPU utilization. Higher GPU utilization allows for the hardware speedup benefit to become more visible.

any of the matrix dimension used in the loss function. One part of the loss function used in HuMoR utilizes the smplx libraries [3] which has a series of rigid body transformations which involves sequentially multiplying many small 4x4 matrices together, and such operation is particularly undesired for FP16 training since the GPU pays a fixed time cost for converting FP32 to FP16 yet the actual computation time is almost nothing, so any speedup cannot be observed.

Comparing Loss Values. We track the training loss and validation loss for training the modified 4xMLP HuMoR model using FP16 and FP32 over 8 epochs. As seen in Figure 5, the training loss and validation loss are closely similar for both FP16 training and FP32 training. This indicates that switching to FP16 training using PyTorch AMP recipe has minimal effect on the training loss values and validation loss values. Since we train with only the CMU dataset due to system constraint and time constraint, our loss value is higher than the loss value in the official HuMoR model.

5. Conclusion

From our investigation, we see that larger models will benefit more using a smaller data format like FP16 for training compare to smaller models. Larger models will do a better job of saturating the GPU to bring GPU utilization higher, and the speedup from using FP16 will become more prominent as the GPU spends more time doing computation. For smaller models, the matrix size is not large enough to saturate the GPU, so the overhead time cost of converting FP32 data into FP16 will become more prominent to the users compare to the speedup from accelerated hardware for FP16.

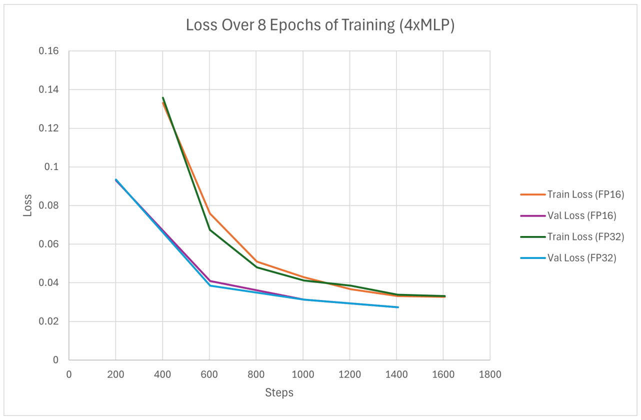

Figure 5: Training loss and validation loss for training the modified 4xMLP HuMoR model using FP16 and FP32 over 8 epochs.

We also investigated the effect of FP16 training on the loss values. From our findings, the loss values for both training and validation are similar between FP16 and FP32 training when using PyTorch AMP recipe.

6. Future Work

This time, we were only able to evaluate the impact of FP16 training on one model and trained for only a short period of time due to our system constraint and time constraint. We learned from this investigation that FP16 training is more beneficial for larger models, so one next step can be to evaluate FP16 training on other existing models that are much larger than HuMoR. Another next step is to obtain a more powerful system with newer GPU and try to train with the entire AMASS dataset and the 2000 epochs used by the original HuMoR model. There is also another smaller format which is FP8. FP8 requires at least Hopper GPU and also has less supporting community compare to FP16, so we didn't look into it this time, but it could be a potential next step. Finally, we only evaluated FP16 training, but it would also be interesting to evaluate the impact of using FP16 for inference.

References

[1] N. Mahmood, N. Ghorbani, N. F. Troje, G. Pons-Moll, and M. J. Black. AMASS: archive of motion capture as surface shapes. *CoRR*, abs/1904.03278, 2019.

[2] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed precision training. *CoRR*, abs/1710.03740, 2017.

[3] G. Pavlakos, V. Choutas, N. Ghorbani, T. Bolkart, A. A. A. Osman, D. Tzionas, and M. J. Black. Expressive body capture: 3D hands, face, and body from a single image. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 10975–10985, 2019.

[4] D. Rempe, T. Birdal, A. Hertzmann, J. Yang, S. Sridhar, and L. J. Guibas. Humor: 3d human motion model for robust pose estimation. In *International Conference on Computer Vision (ICCV)*, 2021.

[5] K. Sohn, H. Lee, and X. Yan. Learning structured output representation using deep conditional generative models. *Advances in neural information processing systems*, 28, 2015.

[6] Y. Wu and K. He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

[7] Q. Zhao, C. Zheng, M. Liu, P. Wang, and C. Chen. Poseformerv2: Exploring frequency domain for efficient and robust 3d human pose estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8877–8886, June 2023.

[8] C. Zheng, S. Zhu, M. Mendieta, T. Yang, C. Chen, and Z. Ding. 3d human pose estimation with spatial and temporal transformers. *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2021.