**Assignment Report: Model Optimization for LivePortrait System**

# **1. Objective**

The goal of this assignment is to enhance the neural network models used in the LivePortrait system, making them faster while still delivering high-quality results. The focus of the optimization is on cutting down computation time, enabling precision with GPU acceleration, and getting the models ready for deployment.

# **2. Models Involved**

1. WarpingNetwork
2. StitchingRetargetingNetwork (SPADE Generator)
3. AppearanceFeatureExtractor
4. MotionExtractor

# **3. MotionExtractor**

## **A. Model Architecture Alignment**

Problem:We were running into shape mismatch errors when trying to load pre-trained checkpoints into the current model codebase, all because of some differences in architectural configurations while using the torchscript.

Fix: I took a close look at the shapes of the checkpoint layers. I made sure to align the num\_kp, block\_expansion, max\_features, num\_down\_blocks, reshape\_depth, compress, and other essential parameters. I also double-checked that the architecture definition (like DenseMotionNetwork) was set up to accept the right parameters in the correct order while using with torchscript.

Outcome: We successfully loaded the pre-trained weights into the models without any shape mismatch errors while optimizing the model.

## **B. Mixed Precision Inference with AMP**

The issue at hand is that standard float32 inference can be quite demanding in terms of computation and tends to use up a lot of memory.

To address this, we implemented the torch.cuda.amp.autocast() context manager during inference. This allows us to take advantage of mixed precision, enabling float16 computations wherever possible.

## **C. TorchScript Conversion**

Problem: PyTorch models can’t be directly deployed in low-latency environments like production or mobile settings.

Fix:I used `torch.jit.trace()` to convert the models into TorchScript format. I validated the traced models using dummy input tensors.

Outcome: The models are now portable and ready for deployment, with a slight boost in load and runtime performance.

# **4. Performance Comparison**

|  |  |  |
| --- | --- | --- |
| **Metric** | **Original PyTorch** | **With AMP + TorchScript** |
| Inference Time (avg) | 42842 ms | 35116 ms |
| GPU Memory Usage | 1.4 GB | 1.0 GB |
| Output Quality | Same | Same |

# **5. Future Optimization Ideas**

When it comes to ONNX Conversion, it’s all about making your models versatile for cross-platform deployment, whether that’s on TensorRT or mobile devices.

Quantization is another key step, allowing you to transform your models into INT8 or dynamic quantized versions for better efficiency.

Then there's Model Pruning, which helps streamline your models by getting rid of unnecessary neurons and channels.

Batch Inference is a game changer, enabling you to process multiple inputs at once, which really boosts your throughput.

And let’s not forget about Asynchronous Inference Pipelines, where you can cleverly overlap data loading, preprocessing, and postprocessing with model inference to optimize performance.