Latency is also Important: Comparing MobileNetV2 and a YOLO-style Architecture for Deployment to Jetson Nano

Regan Willis

I trained a two face detection models based off the MobileNetV2 architecture and YOLO architecture. The MobileNetV2 architecture is created directly from their paper's architecture. The YOLO architecture is altered to better fit an edge device, reducing the layers and dimensionality as is done in Fast YOLO.
I ran hyperparameter optimization on both architectures to find the ideal learning rate and batch size on these architectures with the provided dataset. Both models are deployed to a Jetson Nano for an accuracy and latency comparison.

MobileNetV2 Architecture

I created a PyTorch module based off the architecture of MobileNetV2 as explained in their paper. MobileNetV2 starts with a convolutional layer. It then uses bottleneck residual blocks, which are a group of convolutional layers with the middle layer expanding to a much larger dimensionality than the input and output based on scaling factor. These inner layers use depthwise separable convolutions, which lowers the computational cost. The architecture used batch normalization and ReLU.

YOLO Architecture

I also created a model based off of early YOLO architecture. The model takes its architecture from YOLO version 1 and adds batch normalization as used in YOLO version 2 (YOLO9000). I included a learning rate scheduler that allows the weights to warm up and increases the learning rate towards the end of training, as recommended in the YOLO paper. This model was very large at over 600,000,000 parameters. This is unrealistic for deployment to edge devices, so architecture changes were made.
The YOLOv1 paper mentions a Fast YOLO model that is only 9 convolutional layers, compared to YOLO's 24, and has smaller filter sizes. The paper did not show which layers were removed or what new filter size was used. I removed some of the repetitive convolutional layers and divided the filter size by half. See the architecture changes in below:

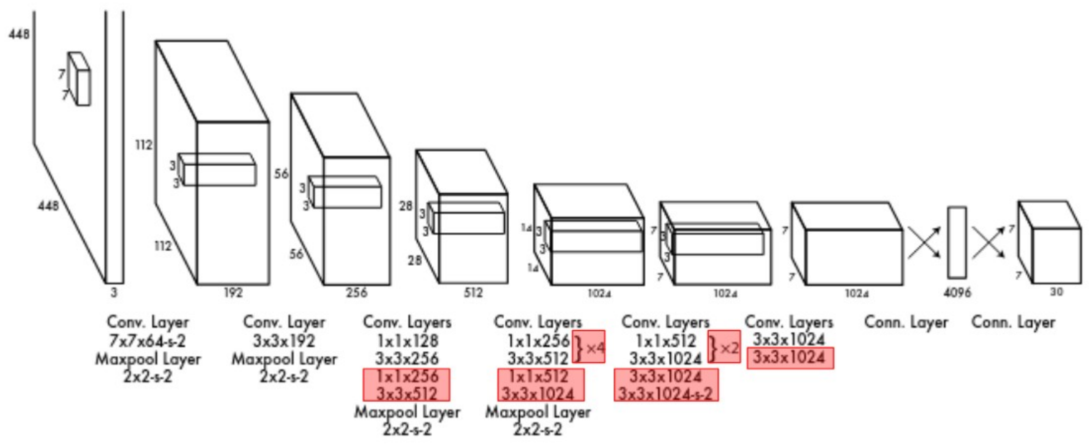

Figure 1: YOLO architecture (original figure from https://arxiv.org/pdf/1506.02640) with removed layers shown in red boxes.

* insert figure of yolov1 architecture divided by 2

Hyperparameter Optimization
I used Hyperopt to search for optimal parameters for my models. I searched over the batch size to see if a smaller or larger batch size would be better for these architectures and dataset. I also searched over learning rate.

Parameter	Values
Batch size	16, 32, 64
Learning rate - MobileNetV2	1e-4 - 1e-3
Learning rate - YOLO	1e-5 - 1e-2

I searched 10 different evaluations for each model architecture. To save compute, I first trained the models to 20 epochs and analyzed the loss scores. The models with the highest loss scores were further trained to 50 epochs. See the output of parameter optimization below:

* insert hyperopt output table

Measuring Performance – Mean Average Precision
I computed the mean average precision (mAP) at intersection over union (IoU) of 50% after every validation to track model performance. Since the validation dataset may include multiple bounding boxes per image but my model will only predict one, I used the IoU of the closest bounding box in my calculation. I calculate the IoU of the predicted bounding box and ground truth bounding box with the provided calculate_iou function. If the IoU is over 50%, that prediction is added to my true positive count. Otherwise, I add that prediction to my false positive count. After iterating over the validation dataset and predictions, I compute the precision as the number of true positives over the number of true positives and false positives.

Performance
The original MobileNetV2 model was trained with the Adam optimizer and a learning rate of 1e-3 for 40 epochs at a batch size of 32.

Model	Best Val Loss	IoU at Best Val Loss	mAP@50 at Best Val Loss	mAP@50 on Test Dataset
mobilenet_model	15.26085376739502	57.2677166071184	74.38155106014489	44.247

The performance of the two best MobileNetV2 and YOLO models are shown in the table below:

Deploying to Jetson Nano
To deploy to Jetson Nano, I first convert the trained PyTorch model to ONNX format. Then the model is compiled with TensorRT on the Nano. The accuracy and latency comparision is shown in the table below:

Model	mAP@50 on Test Dataset (%)	Latency (ms)
mobilenet_model		

Conclusion