An End-To-End Pipeline for Pose Estimation in Mobile Applications Using Kalman Filtering

Finn Dayton
Department of Computer Science
Stanford University
finniusd@stanford.edu

JD Kelly
Department of Electrical Engineering
Stanford University
jdkelly@stanford.edu

Eric Werner
Department of Computer Science
Stanford University
ewern@stanford.edu

Abstract

Pose estimation is a well-studied problem, but running an advanced model on a local device or desktop can have unsatisfying results. Speed is the main issue. In this paper, we investigated using three different models for an end-to-end pipeline application that starts on a phone, goes to a cloud computer running a Flask server to run the model, which sends the results back to the device. We additionally investigated using denoising and a Kalman Filter and report the results. We demonstrate that a successful end-to-end application is feasible if certain device and internet speed latencies are considered.

1. Introduction

Pose estimation is a fundamental task in computer vision that involves estimating the pose or position of an object or human body in a given image or video. Accurate pose estimation is essential in various applications such as robotics, autonomous vehicles, augmented reality, and human-computer interaction. The goal of pose estimation is to estimate the 3D coordinates of key points or joints on the object or body, which can then be used to infer the pose or motion. Pose estimation has been extensively studied over the past few decades, with significant progress achieved due to the availability of large-scale datasets, advances in deep learning, and the development of more sophisticated algorithms.

Before the Deep Learning era, pose estimation was presented as a tree-structured or graphical model problem that predicted keypoint locations based on hand-crafted features [2]. Since the popularization of convolutional deep neural networks, however, results on virtually all datasets has reached new SOTA performance [11] [14].

We investigated three models. First, was the model we presented in our baseline based on [14]. The second was a Pytorch model based on imported ResNet18 weights. The third model is based off [6]. The most advanced of the three, it is a combination of MobileNet V2 [12] and Shufflenet v2 [10]. All three datasets are trained on the MPII dataset [1]. One of the favorable points of the third model is MobileNetV2 has 3.91 million parameters and ShuffleNet V2 has 2.92 million parameters while Resnet18 has 12.26 million parameters. Additionally, MobileNet and ShuffleNet take up 0.49 and 0.31 Giga flops, respectively, while Resnet18 takes up 1.64 Gigaflops. For the mobile application introduced above, a smaller model means lower latency and thus better user experience. In the results section, we compare how the models compare in accuracy of pose prediction.

2. Related Work

Pictorial structures revisited: People detection and articulated pose estimation 2009 [2], Andriluka et al proposes a pictorial structures framework to pose estimation. They show that the right selection of components for both appearance and spatial modeling can lead to SOTA results. The model does not use a deep learning framework and instead relies on handcrafted features.

In Stacked Hourglass Networks for Human Pose Estimation 2016 [11], Newell et al introduce a convolutional neural network architecture for pose estimation. Features of an input images are fed into a neural network architecture that uses a successive steps of pooling and upsampling to achieve SOTA results.

Cascaded Pyramid Network for Multi-Person Pose Estimation 2018 [5] Chen et al introduced a improved architecture over the hourglass model. Unlike the refinement strategy like stacked hourglass, it concatenates all the pyramid features rather than simply using the upsampled features at the end of hourglass module. The model achieved new SOTA performance.

In Simple Baselines for Human Pose Estimation and Tracking 2018 [14], Xiao et al, propose a simpler architecture than either Stacked Hourglass or CPN. The authors use a few deconvolutional layers added on a backbone ResNet-152 and a mask R-CNN [7] over the last convolutional stage. This combined with basic optical flow produced SOTA results for a model much simpler than the prior two. We chose to based our first model on this, and this is what we presented in our project milestone.

There have been several studies on denoising algorithms for images. In Fast Non-Local Algorithm For Image Denoising 2009 by Karnati et al. [8] present improvements to Non-Local Means (NLM) image denoising. Their technique uses modified multi-resolution based approach with much fewer comparison than the original Baudes NLM algorithm [4], making the result nearly 80 times faster. We use denoising in all three of our models.

MobileNetV2: Inverted Residuals and Linear Bottlenecks 2019 by Sandler et al [6] introduces an improved mobile model for object detection. Mobile models are lighter weight models meant to be run on devices with limited compute, e.g., mobile phones. This is exactly our use case. For comparison, MobileNetV2 has 2.11 million parameters versus ResNet101's 58.16 million parameters. They demonstrated MobileNetV2 outperforms state-of-art realtime detectors on COCO dataset [9], used for pose estimation, both in terms of accuracy and model complexity. With these benefits in mind, we use this network within our final model.

In ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design 2018 by Ma et al [10], the authors argue neural network architectures only use complexity (FLOPs) as the metric for comparing models with similar performance. The authors argue that what is needed instead is a measure of model ¬speed. They provide guidelines for designing models on GPUs and CPUs and provide a model ShuffleNetV2 which can process more than double the number of images per second that MobileNetV2 can on a CPU. We also use the design recommendations from this paper in our final model.

Lastly, MobileHumanPose: Toward Real-Time 3D Human Pose Estimation In Mobile Devices 2021 by Choi et al. [6] introduces a mobile-friendly pose estimation architecture. Built on a MobileNetV2 backbone. Their model also incorporates learnings from ShuffleNetV2. It is seven times smaller than Resnet50 and reduces inference time to 12.2ms on a Galaxy S20 CPU. We based our final model on this model but added de-noising and Kalman filtering on top of it.

3. Model Architecture / Approach

We explore three primary model architectures:

1. Baseline model: Resnet18 + denoising
2. Improved Model: Resnet 18 + denoising
3. Current Model: MobilePose + denoising + KF

As the intended model is expected to run on live camera feed from mobile phones, we expect the input images to be noisy. For all models we perform Fast Non-Local Means denoising to reduce noise in input images. This method updates pixel values in a image with the mean of similar patches of pixels in the image to that of the target pixel. This method is very fast, and thus does not significantly contribute to compute time. We saw limited performance gains while testing as the MPII dataset was generally well processed and free of noise.

We begin with an initial implementation to establish baseline performance. Our first model was implemented using Tensorflow. It fine-tuned a ResNet18 base model. The final fully connected layer was replaced to output 32 regressions which mapped to the x and y values of each of the 16 annotated joints.

Building on this work, our second model re-implements a similar architecture in Pytorch, proving more fine control over the training process and is consistent with the framework for most leading pose estimation models. We further increase the size of the fully connected network to map the ResNet18 image embeddings. Outputs reamin directly mapped from ResNet embeddings to the 32 labels consistent with the x and y values of the predicted joints through dense layers.

In initial efforts we develop our architecture for this task,

however, as we discuss, Pose Estimation is a well studied problem. In future model exploration we fine-tune existing light weight models for pose estimation. Our latest model is the MobilePose Model which fine-tunes MobileNetV2 by replacing the final layer with a fully-connected Conv2d() layer, which builds a heat map of size (n/4,n/4) by predicting the probability that each 4x4 square of pixels in the image contains each of the 16 joints. Then, the heatmap for each joint is used to evaluate the most likely candidate coordinates for the joint, which are then fed to the next step.

After the 2D-pose estimation, we feed the observed coordinates (output of fine-tuned mobilenetv2) into a Kalman filter (KF), with the objective to add fluidity (smoothing) and accuracy to predictions on videos (temporal sequences of images).

The KF uses a combination of a prediction step and an update step. The prediction step allows us to instill our pose estimation with a dynamical model, and the update step takes a linear combination of the prediction and the update based on how confident Mobilenetv2 was.

The KF is an iterative estimation technique which uses a combination of a prediction step and an update step. The prediction step allows us to instill our pose estimation with a dynamical model, and the update step takes a linear combination of the prediction and the update based on how confident Mobilenetv2 was. To start, we store our 16 joint coordinates for a timestep as a vector x_t of length 32 (x_i and y_i per joint i) $x_t = [x_1, y_1, x_2, y_2, \cdots, x_{16}, y_{16}]_t^T$. Our KF stores the current and previous coordinates,x_t and x_{t-1} in a vector μ_t. To predict, we multiply a matrix A containing the dynamics model. The dynamics model assumes no acceleration over the timestep, so $\hat{x}_{t+1} = x_t + (x_t - x_{t-1}) = 2x_t - x_{t-1}$, which means the dynamics jacobian is $A_t = [-1 * \mathcal{I}_{32}, 2 * \mathcal{I}_{32}]$. Our covariance is then $\hat{\Sigma}_{t+1} = A_t \Sigma_t A_t^T + Q_t$, where Q_t is the covariance (uncertainty) of the dynamics model.

To output from the KF, we must update our predictions based on the observation z_{t+1}, where the observation is itself the CNN's predicted pose. The output x_{t+1} is a linear combination of the predicted pose and the observed pose, $x_{t+1} = \hat{x}_{t+1} + K_{t+1}(z_t - C_{t+1}\hat{x}_{t+1})$, where K_{t+1} is the Kalman gain. The Kalman gain is $K_{t+1} = \hat{\Sigma}_{t+1} C_{t+1}^T (C_{t+1} \hat{\Sigma}_{t+1} C_{t+1}^T + R_{t+1})^{-1}$. C_{t+1} is the observation jacobian, which is just the identity matrix because the observation is in the same format as the prediction. $R_{t+1} \in \mathcal{R}^{(32,32)}$ is the covariance(uncertainty) in the observation.

We tried multiple approaches to the dynamical model used in the prediction step.

Once we had developed a functional model for pose estimation from images, we focus on an end-to-end pipeline for performing and serving pose estimations as seen in 1. Our intended target is a mobile application. To increase adaptability of the pipeline we perform development in react-native so that the application can run on IOS and Android devices.

The first stage of the pipeline captures live images using a native camera library from the mobile phone's camera. This image is then sent using a REST API to a python Flask server hosted on an Heroku Web app.

The Flask server received the API call, pre-processes the image from the camera, and then feeds it into the pose estimation model. Estimated pose as an array of the x and y values for the 16 predicted joints are returned as the response to the API call. By implementing the model on cloud infrastructure we are able to use all mobile devices are able to leverage this approach, and the pipeline can be updated with the pace of the field to implement state of the art models as they are developed. Leading research has been developed with the Pytorch framework which the current pipeline implements. The cloud infrastructure can also be easily scaled to improve speed and handle larger models.

The app then re-scales the estimated joints and draws the predicted joints over the camera view. This is repeated as fast as latency and compute limitations will allow to provide live pose estimation.

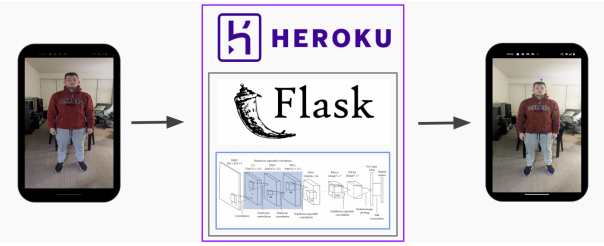

Figure 1. Model Diagram

4. Experiments

In order to establish baseline performance we train the initial model. We partition the MPII dataset for 60% training, 20% validation, and 20% test. Notably, there is only one fully connected layer after the ResNet18 image embeddings which map directly to the 32 outputs. We train with an Alex optimizer, learning rate of 1e-3, and do so for 100 epochs.

In order to improve this model we reimplement a similar architecture in PyTorch. This provides more control over training and is consistent with frameworks of learning models in the field. After iteration on model architecture, we found the best performance came from deepening the network after the image embeddings from the ResNet18 base model. We add two additional hidden layers of size 32 and 128 respectively with ReLu activation functions before the 32 node output layer. Additionally we found the model converges quickly, and train for fewer epochs to reduce over-fitting. Using the same testing split we train with an Alex optimizer, learning rate of 1e-3, and do so for 20 epochs.

After this performance, we choose to implement an existing pose estimation model to drastically increase model performance. It is expected to see considerable performance gains in shifting to a far larger and complex model than those explored previously. Additionally, we recognize that the pose estimation task we plan to leverage the model for is largely based on sequential states for an individual, we see performance could benefit from smoothing introduced by a Kalman Filter and thus implement one for the latest model.

In order to improve the pose prediction pipeline, we conduct further experimentation to increase speed of prediction.

The first speed bottleneck we identify is the rate at which the application is able to extract image data from the live camera feed. As mentioned, the application is developed in react-native to enable cross-platform use, but this high level is particularly slow at accessing low level data. After evaluating several methods, we found implementing native libraries that ran on mobile devices allowed for significantly faster image acquisition, our application implements the react-native-vision-camera library which has native code for cameras on ios and android devices.

Once the image data is extracted we must send the image to the server with a REST API call. Here we found we could significantly reduce image quality without noticeably impacting model performance. This is likely because the model already resizes images prior to estimation, so reducing quality at source does not lose any additional information the model would otherwise leverage.

Another valuable area we determine could be used to improve pipeline speed was the cloud compute resources. The initial server was hosted on the lowest tier Heroku server. We tested pipeline speed with different quality servers to verify this would improve pipeline performance. We saw this scaled with quality which we find is a very reasonable result. This verifies that this framework can scale to reduce latency imposed by compute, and is primarily constrained by network latency.

5. Results

Our initial implementation trained on the MPII test set achieved a baseline Mean Squared Error of 2178.9. This translates to a mean pixel distance from annotated points of 11.7 pixels over the 16 predicted joints.

We implement further improvements on the mode as discussed in our experimentation and similarly train the new model. The training loss can be seen in Figure 2

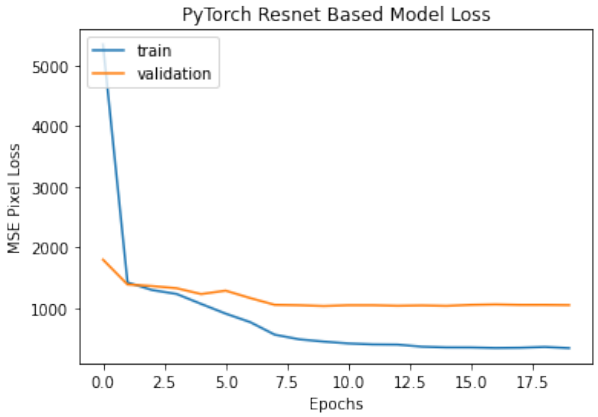

Figure 2. Pytorch Model Training

When tested on the test set, the model achieved a Mean Squared Error of 1569.4. This translates to a mean pixel distance from annotated points of 9.9 pixels over the 16 predicted joints.

Our last model sees significant performance gains as we leverage the existing MobilePose Model. The mean pixel distance from annotated points over the 16 predicted joints reduced to 0.092. This performance gain is consistent with expectations of far superior performance. As stated earlier, our final model uses a heatmap to predict the keypoints as showin in figures 3 and 4.

Figure 3. The input image to the Heatmap

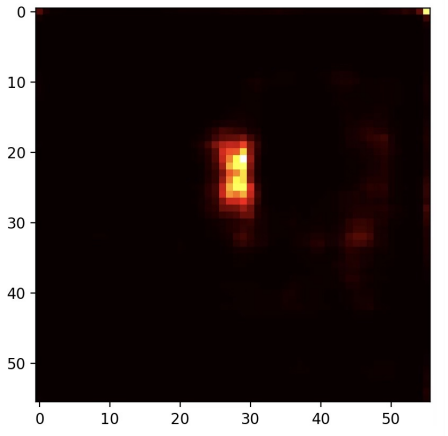

Figure 4. Heatmap Result of the ”head” Keypoint

Furthermore, we recognize the model is intended for use in sequential predictions. We implement an Kalman Filter in order to improve predictions for this kind of pose prediction task. In evaluation, we notice this considerably improves visual the smoothness of the pose predictions in the application.

Once implemented we are able to run the full application prediction pipeline with server set up. See figures 5 6 and 7.

However, the initial pipeline has significant latency. our experimentation is able to drastically improve speed of pose estimation in the application.

Initial image data acquisition time took over 500 milliseconds with high-level react native libraries. However, implementing the vision camera library that leverages native code on android and ios devices was able to reduce acquisition time on our test device (a Pixel 5) to 160 milliseconds.

We further identify that image quality contributed to image data access time. As our model downsizes input images before performing prediction, we are able to significantly reduce image quality before sending to the server without impacting performance. Image acquisition time lowered with image quality to 80 milliseconds where we did not continue to see speed gains, which suggests other overhead that block further reductions.

Figure 5. Front Arms Down

Figure 6. Front Arms Up

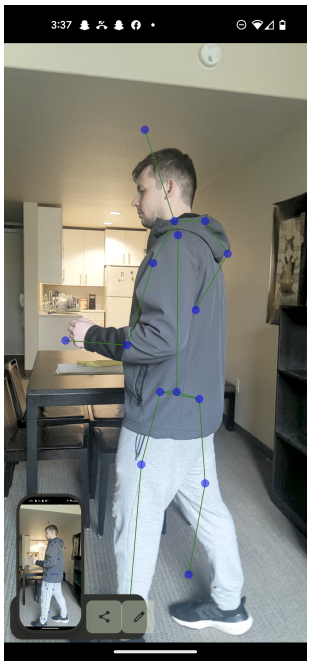

Figure 7. Side View

By further addressing compute time by increasing available Heroku server quality, we are able to achieve a 50 millisecond reduction in model computation time. This indicates that we can further scale performance with cloud compute resources. Here the only hard limitation is the network latency.

Thus the last limitation is the speed of light. While it is unclear if it is possible to exceed this or not given our current understanding of elementary physics [13] [3], we did not consider this for our project. We leave this as an exercise for the reader.

6. Conclusion

In this paper, we focused on evaluating the strategies for an end-to-end pose estimation app. We also looked at the results of implementing and using a Kalman filter. This demonstrates the potential of combining established machine learning techniques with filtering algorithms to further enhance the performance of pose estimation systems in heatmap settings. Further research could explore enhancing the filtering further to include comparing the Kalman filter with the Unscented Kalman Filter, Particle Filter or Recursive Bayesian Estimation.

7. Code Repositories

Github Repository for heatmap of leftApplication: https://github.com/jdkelly199/PoseApp

Github Repository for Flask Server: https://github.com/jdkelly199/PoseServer

Github Repository for MoblePose Model w/ KF: https://github.com/ewernn/MobilePose

Shared Google Drive with Colab Notebooks and Data: Link

References

[1] Mykhaylo Andriluka, Leonid Pishchulin, Peter Gehler, and Bernt Schiele. 1

[2] Mykhaylo Andriluka, Stefan Roth, and Bernt Schiele. Pictorial structures revisited: People detection and articulated pose estimation. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1014–1021, 2009. 1

[3] anonymous. Quantum entanglement communication. 6

[4] A. Buades, B. Coll, and J.-M. Morel. A non-local algorithm for image denoising. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 60–65 vol. 2, 2005. 2

[5] Yilun Chen, Zhicheng Wang, Yuxiang Peng, Zhiqiang Zhang, Gang Yu, and Jian Sun. Cascaded pyramid network for multi-person pose estimation, 2018. 2

[6] Sangbum Choi, Seokeon Choi, and Changick Kim. Mobilehumanpose: Toward real-time 3d human pose estimation in mobile devices. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 2328–2338, June 2021. 1, 2

[7] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn, 2018. 2

[8] Venkateswarlu Karnati, Mithun Uliyar, and Sumit Dey. Fast non-local algorithm for image denoising. In *2009 16th IEEE International Conference on Image Processing (ICIP)*, pages 3873–3876, 2009. 2

[9] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015. 2

[10] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design, 2018. 1, 2

[11] Alejandro Newell, Kaiyu Yang, and Jia Deng. Stacked hourglass networks for human pose estimation, 2016. 1, 2

[12] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019. 1

[13] Ethan Siegel. Even with quantum entanglement, there's no faster-than-light communication. 6

[14] Bin Xiao, Haiping Wu, and Yichen Wei. Simple baselines for human pose estimation and tracking, 2018. 1, 2