Real Time Human Pose Estimation for Boosted Random Forests and Pose Machines

Kenneth Marino, Georgia Institute of Technology

***Abstract*— Current state-of-the-art for human pose estimation is in that it is unsuited for real-time performance without the addition of depth information, which can be a major limitation. In this paper, we extend the work on Pose Machines using GPU acceleration to achieve performance in real time. We also examine and propose solutions to the memory and time issues related to the training of Pose Machines with large datasets. These include a GPU accelerated algorithm for training Pose Machines and changing the way data is used, separating training data into "structure points" and "evaluation points." Finally, we examine the effect of these changes to the speed of testing and training.**

***Index Terms*—computer vision, inference machines, machine learning, pose estimation.**

I. Introduction

A. Problem

The essential problem of human pose estimation from still images can be described simply as the identification of the locations of a number of joints on a two dimensional image. For instance, in our system, we predict the location of the forehead, the base of the neck, the left and right shoulders, the left and right elbows, the left and right wrists, the left and right hips, the left and right knees, and the left and right ankles. An example pose can be seen in Figure 1.

We further specify that we do not have access to depth information. The Microsoft Kinect, which is currently the most robust real-time human pose estimator, collects a depth image using an infrared sensor, as well as an RGB image to do pose estimation [1]. While this allows for efficient prediction of human poses, it imposes a limitation in hardware. Most significantly, it makes pose estimation impossible for outdoor scenes where infrared sensing is ineffective. It also requires the use of specialized hardware which can limit its potential uses.

Another challenge of the problem is to do this prediction in real time. For example, we might like to take a live video stream and overlay part locations so that we can have a real time 2D pose for human subjects. This requires that prediction take place relatively quickly. The Kinect runs at approximately 30 frames-per-second, meaning that part prediction takes no more than about 33.3 milliseconds. Most current approaches to the problem with still images use a graphical model to capture relationships between parts. However, many of these approaches suffer from either accuracy problems because the models are too simple, or suffer from tractability problems because the model is too complex [2]. Clearly time complexity is an issue because inference on these models takes orders of magnitudes longer than what is required for real-time performance.

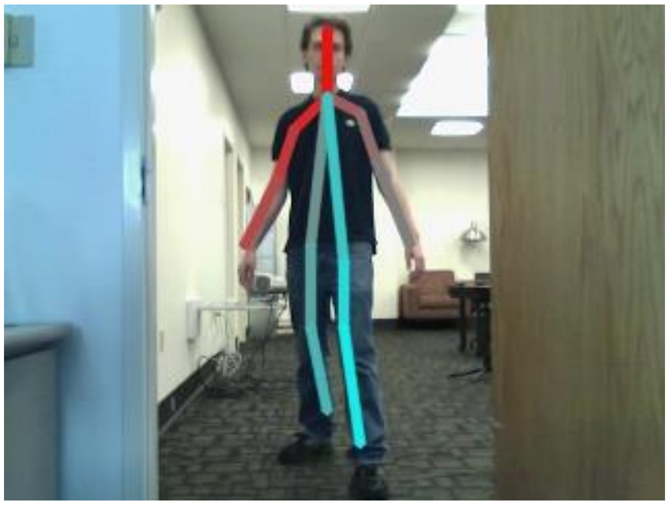


Fig. 1. An example generated pose. The colored line segments show the connection between the 14 annotated locations. The output pose gives the location of joints from which limbs can be inferred.

B. Previous Work

Until quite recently, most work on pose estimation from single images was based on using graphical models to capture dependencies between parts for prediction [3, 4, 5, 6, 7, 8], typically with simple tree or star-structured models. The problem, however, with simplified graphical models is that they do not capture a number of important dependencies, such as symmetric parts (to avoid double counting) and these methods often fail when certain parts are occluded in the image. Inference on exact graphical models is too difficult and computationally expensive, except for very simple models.

More recent work has examined using Deep Neural Networks to train and refine joint predictors [9]. Its use of refining predictions is similar to Pose Machines.

C. Pose Machines

Our approach is to extend the work of Ramakrishna et al on Pose Machines. Pose Machines are currently the state-of-the-art for single-image human pose estimation. This approach sidesteps the issue of representation by approaching pose estimation as a structured prediction problem. The prediction

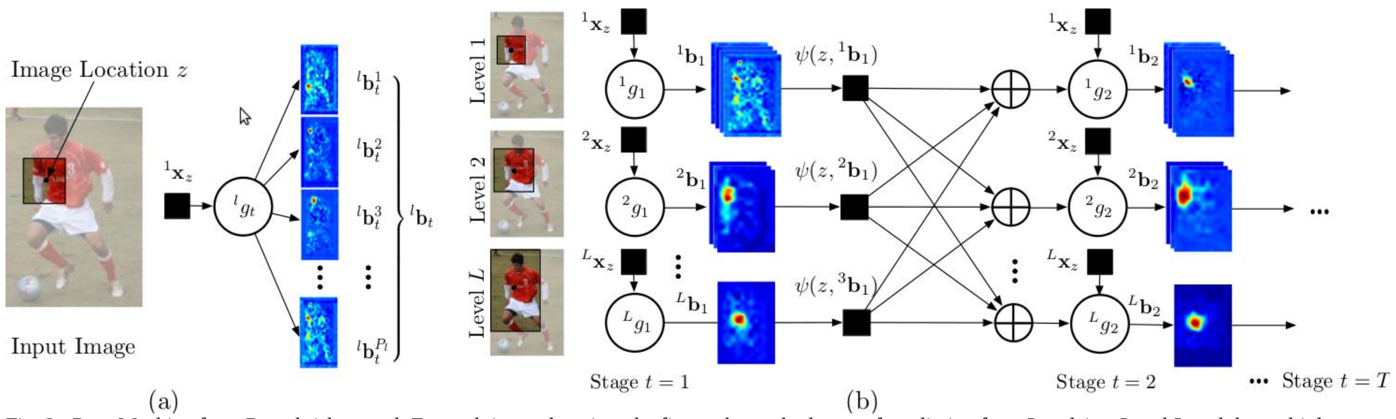


Fig. 2. Pose Machine from Ramakrishna et al. For each image location, the figure shows the layers of prediction from Level 1 to Level L and the multiple stages of prediction from 1 to T. Photo courtesy of Ramakrishna.

task is given an input image, we would like to predict the anatomical landmarks $Y = (Y_1, \dots, Y_P)$ for each of the P parts, in this case P=14 [2]. We predict each part such that $Y_p \in Z \subset \mathbb{R}^2$, where Z is the set of all pixel locations (u, v) in the image. The inference machine then consists of a sequence of multi-class classifiers, $g_t(.)$, which are trained to predict the location of each part. In each stage $t \in \{1, \dots, T\}$, we predict a confidence for each output assignment $Y_P = z, \forall z \in Z$ based on the input image data $x_z \in \mathbb{R}^d$ and contextual information from the previous stage, $\psi(z, b_{t-1}^i)$, where b_t^p represents the confidence at stage t that $Y_p = z, \forall z \in Z$. We can then compute the confidence that a particular part p belongs at location z by $b_t(Y_p = z) = g_t^p\left(x_z; \bigcup_{i=1}^{P} \psi(z, b_{t-1}^i)\right)$, where the union operator is the concatenation of the outputs of $\psi(.)$ for each part.

The intuition behind this framework is that passing information between predictors for each stage allows for information about the location of different parts to be used in the next stage to predict other parts. For instance, a high likelihood of the head in one location might make it far more likely that the neck is at a location near it. This framework, we believe, implicitly captures the statistical relationship between these parts.

The predictors themselves are implemented as gradient boosted random forests classifiers. The basis of this algorithm is the random forest [10]. Given an input $x \in \mathbb{R}^d$, the algorithm returns the corresponding continuous output $y \in \mathbb{R}^P$. This output is determined in the following way. For each tree in the forest, the algorithm determines the leaf node that corresponds to a particular x_i. Each internal node keeps track of a dimension of x dim and a threshold $thresh$. The value of x_i is compared to the threshold at that dimension to determine if the corresponding leaf node is a left or right descendant of the current node – if $x_i[dim] \leq thresh$, it updates the current node to be the left child. Otherwise, it updates it to be the right child. Once it has reached a leaf node of the tree tr, it outputs the distribution of y that is stored there $P_{tr}(x_i)$. This distribution corresponds to the distribution of training examples that also navigated to this node during training. Once this is done for all trees, the final output y is simply the average of all of the $P_{tr}(x_i)$'s.

D. *Fundamental Time, Energy Tradeoff*

One of the fundamental trade-offs in many problems is between speed and accuracy. As one might expect, methods of pose estimation that are more accurate often take longer to compute. This is no different for Pose Machines. Below is a chart showing the major parameters of the Pose Machine Algorithm, how the value of that parameter effects the runtime of the algorithm and the value used subsequently to compare running times for this paper. Increasing each of these parameters will improve the accuracy of the pose estimator to some extent, but causes the prediction to take longer [2].

TABLE I
TUNABLE PARAMETERS

Parameter	Time Relationship	Value in System
Stages	Linear	3
Levels of Hierarchy	Linear	3
Depth	Sublinear	12
Number of Trees	Linear	20
Boosting Iterations	Linear	25
Image Resolution	Quadratic	360p

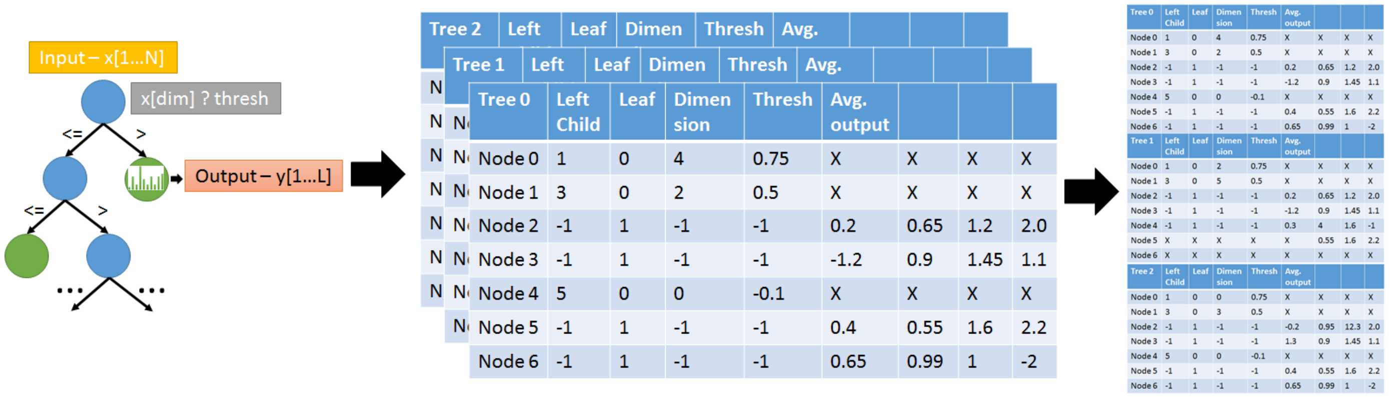
II. APPROACH

A. *CUDA Accelerated Run Time Performance*

Before work was started, running Pose Machines for a complete image took approximately 270 milliseconds using the parameters from Table I. This would mean that you could run at approximately 4 frames-per-second on a top-of-the-line consumer processor. Perhaps additional improvements in processors could bring this up to a reasonable frame-rate, but for now it is simply too slow to allow for real time applications.

A major focus of this work, then, is to increase the speed of this prediction without sacrificing accuracy. To that end, Pose Machines was accelerated using GPU acceleration.

The original work on GPU accelerated algorithms for random forests was performed by Toby Sharp at Microsoft Research. The basic concept of the algorithm was to take advantage of the parallel capacity available with graphics processors and compute the output of all of the inputs x_i, at the same time. Thus instead of computing the output $y_i = f(x_i)\ \forall i$ serially, they are computed in parallel. This is particularly useful with respect to pose machines because we are essentially



Tree 0	Left Child	Leaf	Dimen sion	Thresh	Avg. output			
Node 0	1	0	4	0.75	X	X	X	X
Node 1	3	0	2	0.5	X	X	X	X
Node 2	-1	1	-1	-1	0.2	0.65	1.2	2.0
Node 3	-1	1	-1	-1	-1.2	0.9	1.45	1.1
Node 4	5	0	0	-0.1	X	X	X	X
Node 5	-1	1	-1	-1	0.4	0.55	1.6	2.2
Node 6	-1	1	-1	-1	0.65	0.99	1	-2

Tree 0	Left Child	Leaf	Dimen sion	Thresh	Avg. output			
Node 0	1	0	4	0.75	X	X	X	X
Node 1	3	0	2	0.5	X	X	X	X
Node 2	-1	1	-1	-1	0.2	0.65	1.2	2.0
Node 3	-1	1	-1	-1	-1.2	0.9	1.45	1.1
Node 4	5	0	0	-0.1	X	X	X	X
Node 5	-1	1	-1	-1	0.4	0.55	1.6	2.2
Node 6	-1	1	-1	-1	0.65	0.99	1	-2
Tree 1	Left Child	Leaf	Dimen sion	Thresh	Avg. output			
Node 0	1	0	2	0.75	X	X	X	X
Node 1	3	0	5	0.5	X	X	X	X
Node 2	-1	1	-1	-1	0.2	0.65	1.2	2.0
Node 3	-1	1	-1	-1	-1.2	0.9	1.45	1.1
Node 4	-1	1	-1	-1	0.3	4	1.6	-1
Node 5	X	X	X	X	X	0.55	1.6	2.2
Node 6	X	X	X	X	X	X	X	X
Tree 2	Left Child	Leaf	Dimen sion	Thresh	Avg. output			
Node 0	1	0	0	0.75	X	X	X	X
Node 1	3	0	3	0.5	X	X	X	X
Node 2	-1	1	-1	-1	-0.2	0.95	12.3	2.0
Node 3	-1	1	-1	-1	1.3	0.9	1.45	1.1
Node 4	5	0	0	-0.1	X	X	X	X
Node 5	-1	1	-1	-1	0.4	0.55	1.6	2.2
Node 6	-1	1	-1	-1	0.65	0.99	1	-2

Fig. 4. GPU-optimized data structure for random forests. Sharp Trees are concatenated on top of each other to allow for additional parallel operations. Operations on the data structure are parallel over examples, as well as over trees.

computing this value for Z, the set of all pixel locations (u, v) in the image. Sharp's algorithm stores each tree as a matrix where each row of the matrix is a node in the tree. A modified version of this matrix is shown in Figure 3.

As before, each internal node stores a dimension and a threshold. Instead of doing a conditional branch to determine the successor node, the modified Sharp algorithm calculates the same comparison $x_i[dim] > thresh$ [11]. Now if the condition was true, the value in memory is 1, otherwise the value is 0. The algorithm then loads the index of the next node by calculating $nextnode = leftchild + x_i[dim] > thresh$. The way the tree is laid out, the left and right children of a node are always next to each other. The significance of doing the calculation this way is that no branching is ever performed. This means that all of these calculations can be done simultaneously for $\forall i$ without forcing the GPU to spin some of its threads, which is what would happen with a $if, else$ construct. Once all of the x_i's reach leaf nodes, the average outputs for all of the trees are averaged. Additionally, since we are in fact running boosted random forests, this is repeated for each boosting iteration and the final output is determined by the normal weighted sum of the outputs for each stage.

One additional change that was made from the Sharp Algorithm (besides the generalization of the conditions) is that the operation is further parallelized over trees (see Figure 4). In addition, while Sharp stores the node information (left child, leaf, dimension, thresh) in the same data structure as the average outputs, our modification puts them in separate data structures. These changes allow for more efficient GPU memory use and more efficient use of available threads.

In addition to the random forest prediction being performed on GPU, most of the operations of Pose Machines was moved onto GPU. The most significant calculation moved to GPU was the calculation of the context features, earlier simplified $\psi(z, b_{t-1}^{i})$. Additional details of context features can be found in [11], but the basic idea is that the outputs from the previous stage for all of the parts are condensed into score maps and fed as input into the next prediction stage. In this case, each of these context features for each input x_i is completely independent, so this was fairly easily parallelized.

The boosted adding of the random forests was moved onto GPU. With all of the major calculations now done on GPU, memory copies between CPU and GPU (one of the most computationally expensive operations) were minimized.

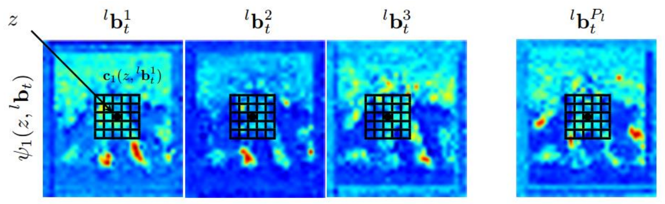

z
${}^{l}\mathbf{b}_t^1$
${}^{l}\mathbf{b}_t^2$
${}^{l}\mathbf{b}_t^3$
${}^{l}\mathbf{b}_t^{P_l}$
$\mathbf{c}_1(z, {}^{l}\mathbf{b}_t^1)$
$\psi_1(z, {}^{l}\mathbf{b}_t)$

Fig. 5 Context Features Visualization. The output values for the locations surrounding each point are concatenated into a new input feature for the next stage of computation. Photo courtesy of Ramakrishna.

B. GPU-Accelerated Training

Runtime performance is an important part of real time human pose estimators, but the other half of the problem, besides the runtime performance, is the training performance. Currently, for a dataset size of one thousand, it takes about 1 day. Unfortunately, to achieve performance close to Kinect accuracy, the algorithm likely needs to train more on the order of 10 million images. Assuming an approximately linear runtime for training (this is likely optimistic). It would take approximately 10,000 days or about 27 years to train a model with 10 million images. Clearly this is intractable.

One approach to fixing this problem is to parallelize using GPU acceleration. For this, first we look at the basic algorithm for training a random forest (the backbone of the prediction, and the part that takes up most of the training time). The recursive algorithm for building a tree, along with the runtime with respect to the number of examples, of different operations is shown in Figure 6.

For parallelization, clearly the most important things to parallelize are the sorting of X, and the determination of the best gain on line 7. There are numerous parallel algorithms for sort, so we will concentrate on the determination of gain.

The determination of the gain basically boils down to the simple operation of determining running sums. The formula we use to calculate the gain for a regression tree is $err_{parent} - (err_{left} + err_{right})$. To calculate a particular error of a particular input dimension and split, the formula is $err = \sum_{j=1}^{D}\left(\sum_{i=1}^{N} w_i y_{ij}^2\right) - \frac{\left(\sum_{i=1}^{N} w_i y_{ij}\right)^2}{\sum_{i=1}^{N} w\_i}$, where N is the number of samples and D is the dimensionality of the output. From this, you can see the bulk of the work is computing these sums. Thus,

```
Algorithm 1: growTree
    Input: $X, Y, active\_indices, active\_features$
1   Compute weighted sum of $Y$ and weighted sum of $Y^2$;
2   Compute parent node info;
3   Shuffle order of features to iterate over;
4   for feat in active_features do
5       $SortX[idx][feat], idxpairsbyvalueofX$;
6       for idx in active indices do
7           $ComputeGain$;
8           $Keepifbest$;
9   for idx in active_indices do
10      if $X[idx][bestFeat] \leq thresh$ then
11          $active\_indices\_left.append(idx)$
        else
12          $active\_indices\_right.append(idx)$
13  if not leaf condition then
14      $growTree(X, Y, active_indices_left, active_features)$
15      $growTree(X, Y, active_indices_right, active_features)$
```

Fig. 6 Algorithm for growing a decision tree. Notice the bulk of the time is spent in the for loop starting on line 4. The running time of this part is $\theta(NlogN)$.

```
Algorithm 2: RunningSum
    Input: $X[1...N]$
1   $height = \lceil \log_2 N \rceil$;
2   $chunkSize = 1$;
3   for idx in active_indices do
4       Parallel: for each set of two chunks C1[1...chunkSize], C2[1...chunkSize] do
5           Parallel: for i = 1 to chunkSize do
6               $C2[i] = C1[chunkSize]$;
```

Fig. 7 New Algorithm to compute running sums in parallel.

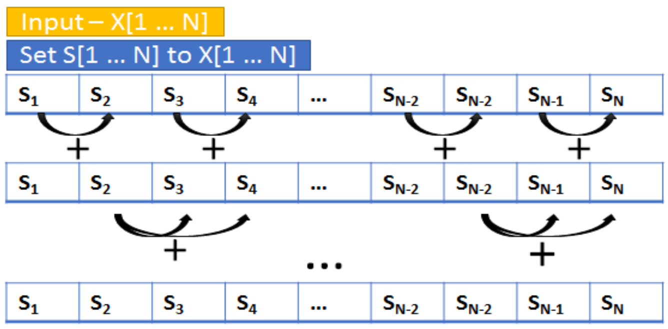

Input – X[1 ... N]
Set S[1 ... N] to X[1 ... N]
S_1 S_2 S_3 S_4 ... S_{N-2} S_{N-2} S_{N-1} S_N
+
...

Fig. 8 Visualization of running sum algorithm in Figure 7.

we developed the following algorithm for computing the running sum in parallel shown in Figure 7 and Figure 8.

The running time of this algorithm is $\theta(\frac{N}{k} logN)$, where k is the number of parallel processes available. The only serial work is related to the log of the size of the input. The adding operations occur in parallel, but because hardware has a fixed limit to the number of parallel operations, the actual time depends on that hardware constant k.

III. Results

Figure 9 shows the runtime performance of Pose Machines using the parameters from Figure 3 using naïve CPU parallelization and the GPU acceleration version.

These results show a great improvement in runtime from about 4 frames per second to about 17, and approximately 400% increase. In particular, the context feature calculation is greatly speeded by GPU acceleration, however, the prediction is still greatly sped up. The only operation not implemented on GPU was the Histogram of Gradients (HOG) feature computation. Clearly speeding up this computation could have a positive effect on runtime performance.

Figure 10 shows the speed results for the GPU accelerated running sum algorithm. For sufficiently large inputs, the GPU accelerated algorithm runs about 3 times the speed. While this result is certainly promising, GPU acceleration of training may not be the ideal solution. First, there is a memory issue in that even high-end GPU processors have only about 10 GB of memory which may prevent GPU from being used for training. In addition, given that GPU memory is filled, only one CPU process can be used at a time during training which prevents CPU acceleration of training. This means that unless the GPU

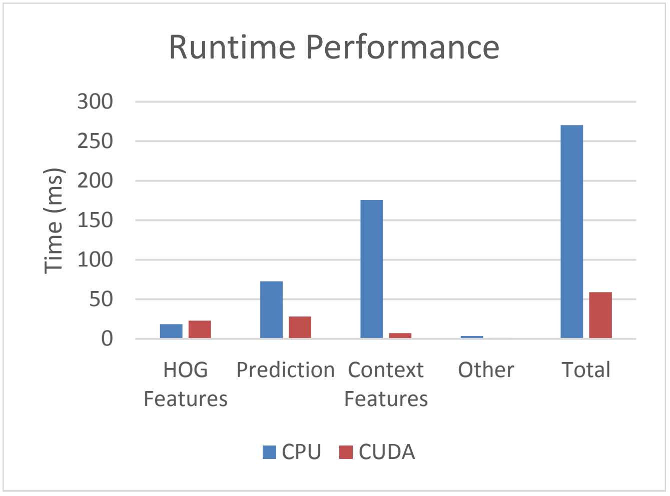


Fig. 9 Bar graph showing runtime performance of Pose Machines. All parameters are listed in Table I.

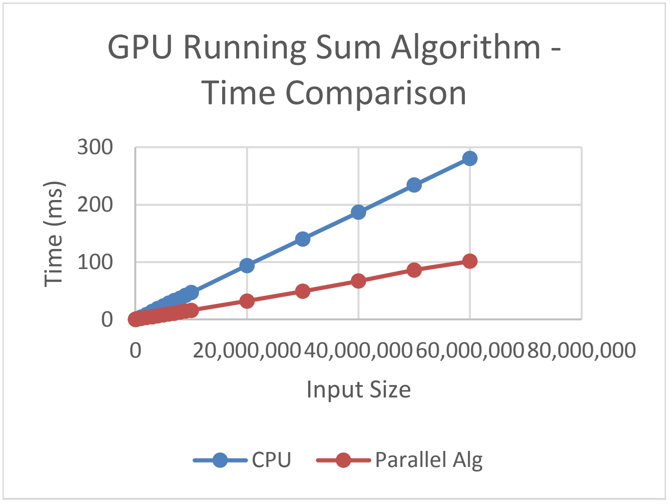


Fig. 10 Line graph showing performance of new running sum algorithm compared to simply calculating a running sum on CPU.

accelerated training is far faster than a single-core CPU algorithm, GPU accelerated training is unlikely to help.

IV. Conclusion

The most impressive result of the work is the time performance. Based on the number reported, running Pose Machines using GPU acceleration achieves performance close to that of the Kinect with respect to speed. It is currently the fastest implementation of a working pose estimator that does not use depth information. DeepPose [9] claims a running time of 100 milliseconds, compared to our 60 milliseconds.

The accuracy of the model is still the biggest problem. Pose Machines still do not match the prediction accuracy of the Kinect. One of the major problems right now is the limited training sets that the algorithm is trained on. Improvements in performance from GPU accelerated training and other improvements in training time such as the structure and evaluation point schema will be important to train the algorithm on extremely large datasets.

V. Future Work

Most of the future work on this project will concentrate on training larger datasets. This will involve solving problems surrounding training speed as well as problems with memory. Currently, the entire training dataset is put into memory for training, but as the dataset gets larger, this will become intractable.

Additional work is currently being done to train by using some input examples as “structure points” and used to determine splits and using the rest to determine the distribution on leaf nodes. Ideally this would reduce the runtime by reducing the number of “splits” that internal nodes have to consider during training. This work is ongoing, so no results have been generated, but the preliminary findings make this a promising area.

A possibility beyond directly solving the training time and space issues is to better use available training examples. The idea is to essentially learn which images will best improve the performance of the algorithm. The algorithm will train on one subset of the data and then predict the output for the other subset of the data. The images that the algorithm did poorly on will then be used in the next round of training so that it can better learn the things it missed.

References

[1] J. Shotton et al, “Efficient Human Pose Estimation from Single Depth Images,” *CVPR*, 2011.

[2] V. Ramakrishna, D. Munoz, M. Hebert, J. A. Bagnell, and Y. Sheikh, "Pose Machines: Articulated Pose Estimation via Inference Machines," *ECCV*, 2014.

[3] P. F. Felzenszwalb, D.P. Huttenlocher, “Pictorial structures for object recognition,” *IJCV*, 2005.

[4] D. Ramanan, D.A. Forsyth, A. Zisserman, “Strike a Pose: Tracking people by finding stylized poses,” *CVPR*, 2005.

[5] M. Andriluka, S. Roth, B Schiele, “Monocular 3D Pose Estimation and Tracking by Detection,” *CVPR*, 2010.

[6] M. Andriluka, S. Roth, B Schiele, “Pictoral Structures Revisited: People Detection and Articulated Pose Estimation,” *CVPR*, 2009.

[7] Y. Yang, D. Ramanan, “Articulated pose estimation with flexible mixture-of-parts,” *CVPR*, 2011.

[8] S. Johnson, M. Everingham, “Clustered pose and nonlinear appearance models for human pose estimation.” *BMVC*, 2010.

[9] A. Toshev, C. Szegedy, “DeepPose: Human Pose Estimation via Deep Neural Networks,” *CVPR*, 2014.

[10] L. Breiman, “Random Forests,” Machine Learning, 2001.

[11] T. Sharp, “Implementing Decision Forests on a GPU,” *ECCV* 2008.