CNN for Joint Location Estimation

Chris Rockwell, Uzziel Cortez, Eric Huang
Department of Electrical Engineering and Computer Science
Department of Applied and Interdisciplinary Mathematics
University of Michigan
Ann Arbor, MI, 48109
cnris@umich.edu, uzzielc@umich.edu, erhuang@umich.edu

1 Problem and Motivation

Much of the technology that is driving innovation across a variety of fields relies on the stable solution to one of the most difficult problems in computer vision and artificial intelligence: being able to discern what activities people are performing from an image. Being able to understand a person's actions serves a range of purposes, especially in fields involving human-computer interactions and for the safety of autonomous driving vehicles, for example. One of the most crucial steps in understanding what people are doing is the estimation of a person's pose, i.e. their orientation, the articulation of their limbs, posture, and the relationships of joints adjacent to one another.
Pose estimation is a fundamental component for understanding the larger-level context of what's going on in an image, beyond the kind of information that can be extrapolated from analytical pixel-by-pixel scrutinies. In the context of detecting human joints, specific challenges include overcoming the variety of postures that the human body can contort itself into, overcoming variation in clothing styles, and dealing with occluded joints.
Our goal to accomplish detection of pose is to more accurately detect joints since we can infer pose from there. We will accomplish this by re-implementing and improving on Newell et al.'s 'Stacked Hourglass' method in PyTorch which will allow us to determine the exact pixel location of human bodily joints in a picture [1]. They accomplish this by generating gaussian heatmaps from ground truth pixel locations, and minimizing mean squared error between these heatmaps and heatmaps generated through their network from the input image. For this non-convex optimization problem, they use the RMSprop optimizer to determine losses and then perform back-propagation to update network weights.
We wish to recreate their model in PyTorch due to PyTorch's versatility and expansiveness in machine learning, and also experiment with a different optimizer called Adam and other conditions to see how our additions would compare to the baseline model created by Newell. We will be using the same methods and evaluation criteria to allow for valid comparisons. When the model outputs gaussian heatmaps of the likelihood of each joint, we will select the maximum value as the location of the joint. We will compare these predictions to the groundtruth and classify the predictions as correct if it is within 0.5 of the person's head size. Given that the main differences in architecture will be the optimizer and batch size, we expect training time for each variation to be similar to Newell's training time of three days, thus limiting how many variations we can make to the model. Depending on the results, Newell's model combined with our modifications could be promising since pose estimation is an important piece in many broader applications (as mentioned earlier), and solving this challenge would open the door for posterity to build upon this work and further the evolution of artificial intelligence.

2 Related Work

Needless to say, there are already many pose estimation methods that have been put forward by others in the fields of machine learning and computer vision. The paper we've looked at includes Table 1, a short anthology of recent evolution of pose estimation performance on the MPII dataset. Clearly

there is room for improvement, as some joints are still being misclassified 10-20% of the time.
Predictably, some joints are much easier to detect than others. It isn't surprising that the head would

Table 1: Accuracy Results on MPII Human Pose dataset[1]

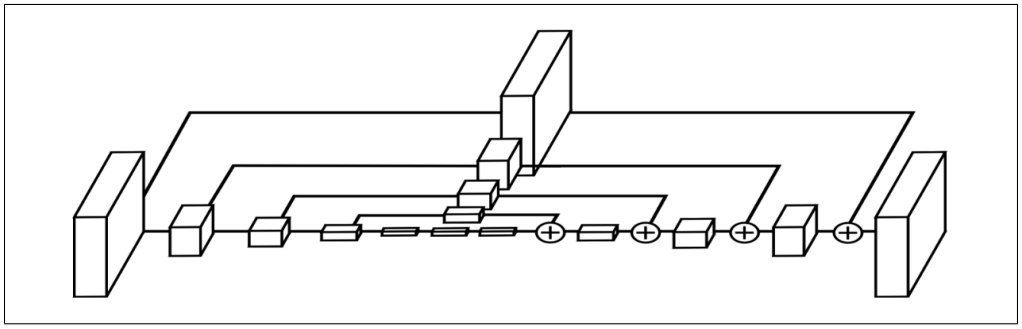
Authors	Head	Shoulders	Elbow	Wrist	Hip	Knee	Ankle	Total
Tompson et al. [11]	96.1	91.9	83.9	77.8	80.9	72.3	64.8	82.0
Carreira et al. [13]	95.7	91.7	81.7	72.4	82.8	73.2	66.4	81.3
Pishchulin et al. [12]	94.1	90.2	83.4	77.3	82.6	75.7	68.6	82.4
Hu et al. [14]	95.0	91.6	83.0	76.6	81.9	74.5	69.5	82.4
Wei et al. [6]	97.8	95.0	88.7	84.0	88.4	82.8	79.4	88.5

be the easiest to detect, since the head area often has plenty of texture and definition available. On the flip side, a joint like the ankle can be difficult to correctly discern because of their small size, and it's much more likely that they become intermingled with other parts of the image or are blocked altogether. As the technology to understand how to process images improves, these numbers will continue to get better and better, and though not perfect, the 'Stacked Hourglass' method is next in line in this evolution and is able to outperform all of the methods listed above.
Newell et al. describe their work multiple times as being highly inspired by the work of Tompson et al. [2]. Tompson et al. implement a sliding window feature as a method of refining their initial predictions, one that moves the network around the image at different scales. They have pointed out that this makes their system invariant to the translation of joints within the image. However, Newell et al. state in their paper that this method not only can be expensive to implement, but includes redundant convolution operations. Furthermore, they argue that this method of refinement is rather ineffective against errors in joint occlusion in the image or to mislabeled joints/limbs.
Another notable difference between the hourglass modules and other designs is the use of symmetry. There have been many other designs that likewise make use of multiple scales to process spatial information [2, 3, 4], including fully convolutional network systems, but these often have unequal treatment of their images across all its scales. The hourglass' symmetric topology treats the high-to-low resolution process equally with the low-to-high resolution process. It's probable that this lack of symmetric has contributed to past systems operating sub-optimally.

3 Methodologies

In the deep learning community it has become standard practice to use convolutional neural networks when working with image data sets[1]. A convolutional neural network makes use of multiple layers including, convolution layers, pooling layers, and upsampling layers to process the visual data and extract important features that are used to solve a task. The architecture used in this model, first proposed by Newell [1], is called 'Stacked Hourglass Network' and is composed of many of these kinds of layers arranged in a specific way such that when visualized appropriately produce a hourglass type topology, see figure 1.

Figure 1: Visualization of Single Hourglass architecture[1]



A convolution layer will take in visual data, and using a filter(s), or kernels, that are stored in the layer, computes a cross-correlation which are called a feature map(s), or activation map(s). The weights

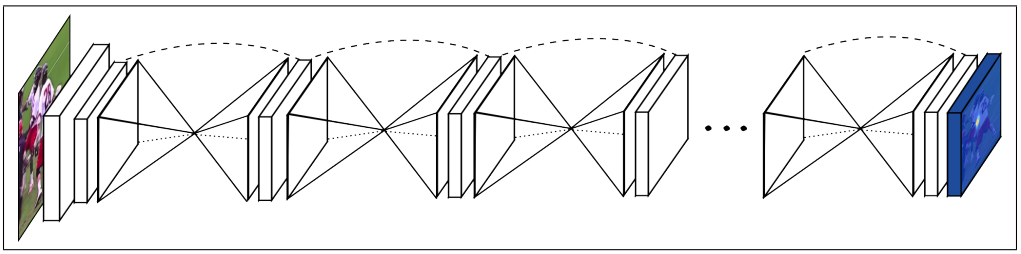
that are used in the kernels are trained to activate on features described by the groundtruth data during the training process [3]. If there are multiple filters in the convolution layer, the activation maps are stacked on top of each other along the third dimension to produce the output volume for that layer. The resulting output volume of a given layer will have lower spatial resolution than that of the data that was fed in as an input into this layer. Each entry in the output volume represents the correlation between the pixels in a window of the input data volume. The data contained in the output volume can be interpreted as an abstract representation of features that are contained in the input data volume. After the output volume of the convolution layer is computed, it is fed in to the next layer called a pooling layer. Pooling is a form of non-linear downsampling of the input data. In our architecture, we use max pooling layers. Max pooling layers will act on non-overlapping windows of this layers input and output the max value in the given window. Again, the resulting output data volume for this layer will be of lower spatial resolution than that of the input. We can interpret the entries in the output of this layer as the 'most important' parts of a feature of the given input data for this layer. The intuition behind the use of this layer is that we care to find what features are important rather than the location of the features. This type of layer also serves to reduce the number of parameters used in the model, as a product of the lowering of spatial resolution, which in turn can help with computational complexity as well as overfitting.

As standard practice, after a pooling layer we would see the use of an activation function such as sigmoid or ReLu. Using an activation function after the convolution and max pooling layer increases the non-linearity of our approximation of the function we are trying to estimate. Also, the ReLu activation function helps alleviate the vanishing gradient problem by having constant slope. This means that during the back propagation phase, if a gradient is to be passed through a layer its derivative is one, if the gradient does not pass through a layer its multiplied by zero and the neuron dies. Thus if gradients must be allowed to pass through they have a clear signal, and if not, sparsity is introduced and we can reduce computational complexity. With other activations functions which are not as efficient, a gradient may be expected be passed through a layer during back propagation, but may be multiplied by a value close to zero thereby slowing down the training process by not sending a clear signal or not killing the neuron [4].

Once a desired spatial resolution is reached by a series of convolutions, max pooling, and activations, we implement a series of nearest neighbor upsampling layers. An upsampling layer will take the input volume and increase its spatial resolution by an integer factor (user specified) and interpolates using nearest neighbor. This allows for the interpolation of features, found by the convolution layers, back into the original spatial resolution of the input volume.

The 'Stacked Hourglass' architecture, see figure 2, can now be described using the layers discussed

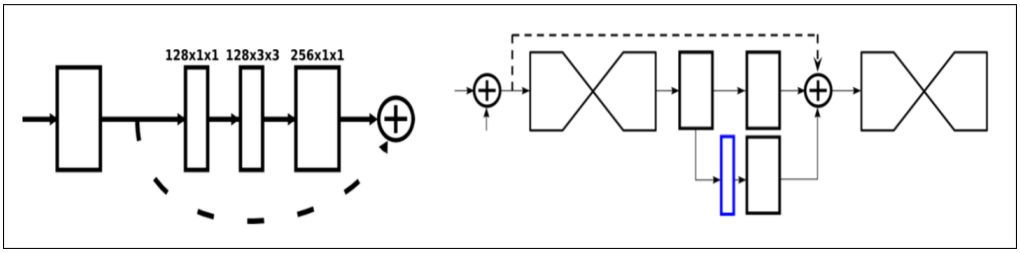
Figure 2: Visualization of Stacked Hourglass architecture[1]



above. We will consider a single hourglass in our description and the model architecture will be a series of single hourglasses stacked sequentially. The first half of the hourglass is composed of blocks of layers. Each block is essentially composed of a convolution layer, max pooling layer, and ReLu activations. As the data makes a forward pass through the hourglass, each block sequentially extracts various features in the image and reduces the spatial resolution as well. After flowing through a series of these types of blocks, we reach a minimum resolution, dictated by the architecture, which concludes the first half of the hourglass. The second half of the hourglass is composed of sequential nearest neighbor upsampling layers and residual connections. Using upsampling, we are able to take the coarse information that has be generated by the first half of the hourglass and interpolate the data back to the original input image size. Residual connections represent element wise addition between the output of an upsampling layer and the output of the convolved layer with matching resolution. These kinds of connections prevent vanishing or exploding gradients during the training process which can be an issue for deeper networks. Additionally, through the use of residual connections

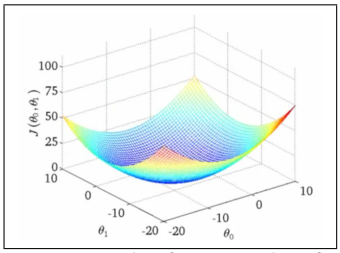
we can incorporate fine and coarse information about the image at prediction times, where the fine information comes from the first half of the hourglass and the coarse information comes from the second half of the hourglass.
The second half of the hourglass is composed of sequential nearest neighbor upsampling layers and residual connections [7]. Using upsampling, we are able to take the coarse information that has be generated by the first half of the hourglass and interpolate the data back to the original input image size. Residual connections represent element-wise addition between the output of an upsampling layer and the output of the convolved layer with matching resolution. These kinds of connections prevent vanishing or exploding gradients during the training process which can be an issue for deeper networks. Additionally, through the use of residual connections, we can incorporate fine and coarse information about the image at prediction times, where the fine information comes from the first half of the hourglass and the coarse information comes from the second half of the hourglass.
The final architecture involves the stacking of multiple hourglass structures on top of each other. This stacking along with applications of the loss function between hourglasses allows the model to be intermediately supervised. Once the model reaches an intermediate loss function a prediction is generated and the model is able to progressively refine its prediction as the data flows through the other seven stacked hourglasses [1], see figure 2.
As we compute the loss between each of the hourglasses to fine-tune our parameters (figure 3), it is

Figure 3: (Left) Visualization of residual module with intermediate supervision. (Right) Intermediate supervision between hourglasses using the selected optimizer to compute the loss function with the ground truth. [1]

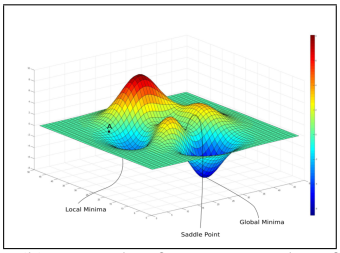


important that we find the minimum of the loss function to optimize our weights of the parameters during back-propagation. One method we learned in class was gradient descent, which is based off of first-order partial derivatives of the loss function. The gradient descent computes the steepest gradient of its sample of the loss function and travels along that path at a fixed learning rate.
However, we learned in class that there are various flaws to the gradient descent method, such as overshooting the local minima of the loss function. If the learning rate is fixed, it's possible for the gradient descent method to continuously oscillate around the local minima and never reach it. We can adjust the learning rate to decay after various iterations, but we then are confronted with the second problem of getting stuck in local minimas.
In strictly convex functions, this is perfectly fine since the local minima is also the global minima.

Figure 4: Example of a non-convex loss function



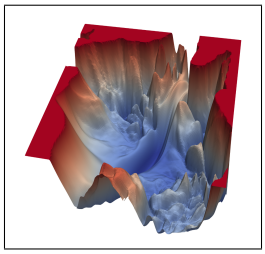
(a) Example of a convex loss function.



(b) Example of non-convex loss function

However, loss functions are never as simple and easy to calculate as in the example shown in Figure 4(a) with the example of just two parameters. In reality, loss functions are incredibly complex with lots of local minimas and maximas as shown in Figure 4(b) (with the example of two parameters).

Figure 5: Highly complex loss function



With complex loss functions like the one in Figure 4(b), the gradient descent methods will not work due to the possibility of getting stuck in local minimas while also getting stuck in saddle points which is common when minimas are continuous with maximas. Not only that, but real problems have hundreds of parameters, and so loss functions become incredibly complex as seen in Figure 5. Saddle points pose an issue for gradient descent method since it could change for one variable but find it a minima for another variable and oscillate within that direction, making us think that it is converging to the minima.

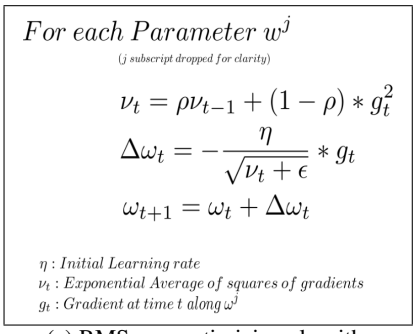
One method we learned in class to overcome these hurdles was the stochastic gradient descent method. Instead of taking the gradient of the summed-up components of the loss function, we take the gradient of each training sample as we find the minima. The convergence behavior as we learned in class is more sporadic but eventually converges to the minima, and these sporadic spikes provide the chance to escape local minimas.

A growing popular idea in gradient descent optimization to resolve these issues is the idea of momentum. By using a fixed percentage rate and multiplying the previous gradient to then add it onto the new calculated gradient, we are able to speed up and slow down the descent as needed, thus helping with oscillations. The momentum term will increase when multiple dimensions have gradients pointing in the same directions, and vice-versa if the gradients change directions.

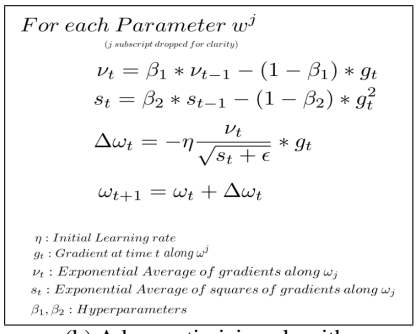
Newell used RMSprop [1], a continuation of the momentum idea, to minimize their loss function [1]. RMSprop is summarized in Figure 6(a), in which three equations are calculated for each parameter [8].

The first equation is calculating the exponential average of the square of the gradient for that

Figure 6



(a) RMSprop optimizing algorithm



(b) Adam optimizing algorithm

parameter. This corresponds to the projection since it is the gradient along the direction of the parameter. The last exponential average is multiplied by ρ while the square of the gradient is multiplied by 1-ρ, allowing us to weigh more recent gradient updates heavily.

The second equation involves calculating the step size using an initial learning rate and dividing it by the average to dampen possible oscillations and move towards the minima more quickly. The third equation is the update step [10].

In the machine learning area, Adaptive Moment Optimization, or Adam, has become the state-of-the-art default optimizer for loss functions. Adam combines the heuristics of both momentum and RMSProp to converge faster towards the minima without being at risk to oscillations [9]. Adam is similar to RMSProp in that it is calculated for each parameter as seen in Figure 6(b). Like in

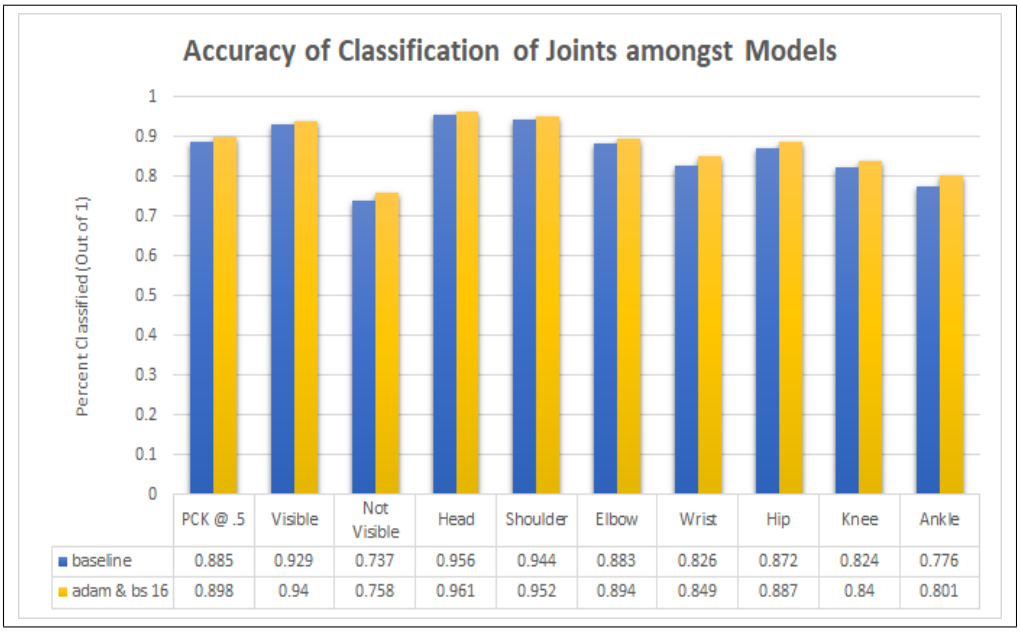
RMSProp, we factor in the previous iteration of the gradients as we calculate the current gradient and square of it in equations 1 and 2.
The main difference is in equation 3 where we divide the exponential average of the gradient by the root-mean-square of the exponential average of square of gradients and multiply them by the initial learning rate and gradient [10].
Due to the promising nature of Adam and how it combines both Momentum and RMSProp, we have decided to make it a hyperparameter and see if our model performs better when we use Adam for our loss function. The right optimization method for our loss function could mean the difference in computing time of hours, and even days, since these loss functions will have hundreds of parameters, if not thousands.
An important factor that can affect the convergence of the optimizer is batch size. By performing the optimizer with multiple samples at once, we can calculate the losses in parallel to leverage parallel processing and speed up computation time. Since the original paper used a batch size of eight, we wanted to experiment with a larger batch size of sixteen to see if it would produce better or worse results. We expect to see better results after increasing the batch size since using more examples to calculate a partial gradient before each update should lead to a more accurate gradient estimate.

4 Evaluation

We run our architecture on the same training set used in the original paper, the MPII Human Pose dataset [5]. The MPII Human Pose is a benchmark for training images due to the 25k images with annotations for multiple people, all in all providing over 40k annotated samples. MPII contains images of people performing a variety of activities, thus giving our architecture a robust set to work with. We increase the difficulty of the data set by including random rotations, flips, scaling, contrast alterations, and brightness alterations.
As mentioned in the previous section, we experimented with Adam as our optimizer versus the original architecture of RMSprop. Adam is becoming popular within the community due to its combination of momentum along with RMSprop, so we wanted to see how our results would fare with Adam as our optimizer. We also experimented with batch size to see if that would increase our accuracy since batch sizes affect the convergence of our optimizers.
Our loss function was kept the same, using mean-squared-error loss to compare the ground truth to the gaussian heat map which is the output of the networks and tune the parameters from there with our optimizers. Our test hardware was 4 Tesla k40 GPUs, while the original paper used an NVIDIA Titan GPU. We first keep the architecture the same and plotted the validated loss over training iterations. We then run the same experiment with the resolution change, then just the batch size change, then just using Adam as our optimizer, and then finally batch size combined with Adam as our optimizer.
As seen in Figure 7, we measure the accuracy of the baseline architecture and our modified architecture with the standard Percentage of Correct Keypoints (PCK), or how many key points we correctly classify. We train our model on the same subset of the training images as Newell's and then run our evaluation on the same validation set of 3000 images from the MPII dataset that Newell used [1]. We classify the predicted joint as incorrect if the distance between the prediction location and the ground truth location half of the subject's head size (PCK@0.5) via the training images' ground truth. In Figure 7, we see that we are able to recreate the results from the original paper since our baseline model is able to get the same results as the original paper (PCK@0.5 was 0.881 while ours was 0.885) [1]. Not only that, but our modified model with Adam combined with increased batch size performs consistently better by about a percentage point across all joints.
When we measure the loss compared to the ground truth of the images, as seen in Figure 8, the model with the only change being Adam as the optimizer, we see a slight improvement over the baseline model proposed in the original paper. However, when we change only the batch size, there is a noticeable improvement in validated loss, and when we have both batch size change along with Adam as our optimizer, that's when the model performs the best.

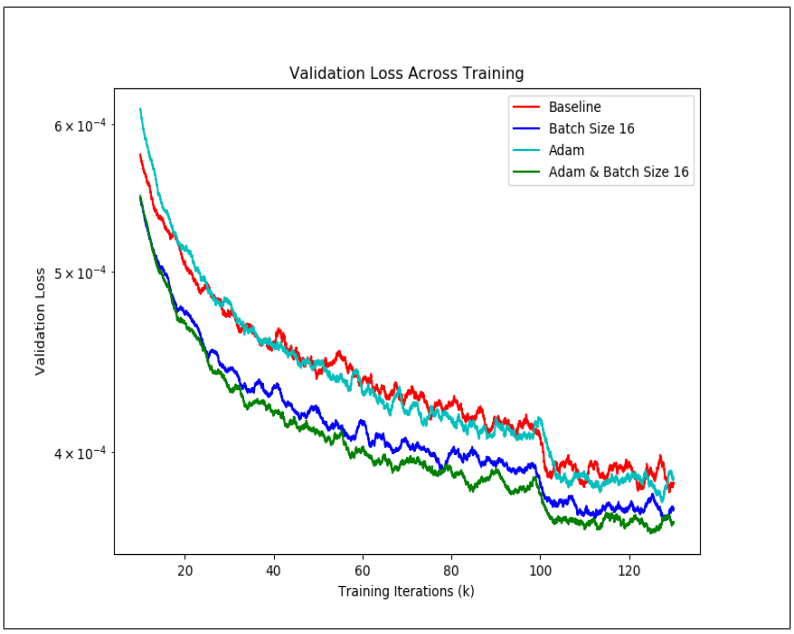
The model outputs are shown in Figure 9(a-d), with white dots being the ground truths, red dots being incorrect predictions from our model, and pink dots being correct predictions from our model. In Figures 9(a,b), it is clear that occlusion affects our results. The occlusion is worst in Figure 9(a) as our model tries to find the joints for the elderly gentleman in the car while being blocked by the man in center. All of the white dots have no pink dots over them, and the model's predictions were

Figure 7: Per joint comparison between Newell et al.[1] results and our proposed model's



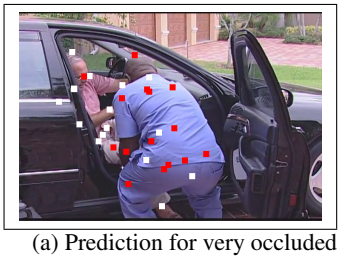
	PCK @ .5	Visible	Not Visible	Head	Shoulder	Elbow	Wrist	Hip	Knee	Ankle
baseline	0.885	0.929	0.737	0.956	0.944	0.883	0.826	0.872	0.824	0.776
adam & bs 16	0.898	0.94	0.758	0.961	0.952	0.894	0.849	0.887	0.84	0.801

Figure 8: Validation comparison of loss between Newell et al.[1] model and our modifications



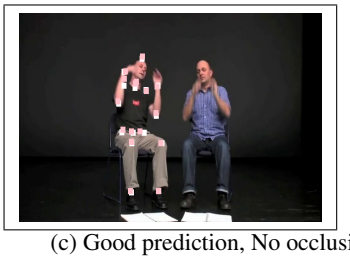
all incorrect red dots. In Figure 9(b), the occlusion is not as bad but still causes our model to have incorrect predictions, specifically the joints (hip, leg sockets) behind the shovel. When the target in the image has no occlusion, our model is able to predict all of the joints correctly as seen in 9(c). Figure 9(d) shows a Gaussian heatmap output from our model, where the lighter color represents a greater likelihood of the joint being located at that pixel.

Figure 9

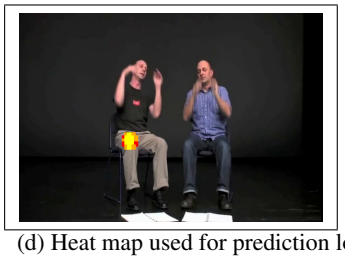


(a) Prediction for very occluded joints

(b) Prediction for partially occluded joints



(c) Good prediction, No occlusion



(d) Heat map used for prediction location

5 Conclusions

After studying Newell's paper, learning PyTorch, and studying the model, we were able to successfully recreate Newell's work while also learning about deep learning, neural networks, optimizers, and image processing. Our results showed that our recreation of Newell's model in PyTorch was successful since it produced a classification accuracy of 0.885 while Newell's model produced an accuracy of 0.881. Not only that, but our provided modifications to the architecture then improved the classification performance across all joints by at least a percentage point via Adam as our optimizer and an increased batch size.

As seen in the results in figure 9, our modifications are still weak to occlusion of test subjects in the images. However, when the subjects in images are visible without occlusion, we consistently outperform Newell's model, showing that Adam as an optimizer along with increased batch size is incredibly promising. Increasing batch size alone or just changing the optimizer to Adm will improve results as seen in figure 8, but combining both batch size along with using Adam as an optimizer performs the best.

The scope of the project proved to be challenging however due to the complexity and scale of the project. The team ran into struggles with learning PyTorch for conversion and also with high training times since each changed model iteration took around 3 days to train, limiting us on how many changes we want to make to the model while still being able to gather results. It was only with early knowledge of this assignment and careful time management along with a relatively early start was the team able to finish this project in time with the long training times.

6 Contributions

Chris had the most experience working with PyTorch and neural networks and so helped the other group members Uzziel and Eric with learning the semantics of PyTorch to help with conversion of the model. With enough tutoring and coding sessions together, the group was able to split up and work on converting various parts of the models provided on GitHub to PyTorch and learning which code corresponded to what part of the architecture for future modifications. We then worked together on analyzing the various parts of the model and deciding which parts of the model to change and see if they can improve upon the model. Due to the staggering amount of images and potentially long training times for each changed variable for comparison, we settled on the changes listed in methodologies and results. Because Chris has access to powerful GPUs thanks to his research lab, Chris ran the training and tests and would collect the results after for the group to analyze. While Chris ran training and tests, Uzziel and Eric worked on the proposal and paper to explain concepts and background. When results were collected, we all came together to analyze the results and finish up the rest of the paper.

References

[1] Newell, A., Yang, K., Deng, J.: Stacked Hourglass Networks for Human Pose Estimation. In: European Conference on Computer Vision (2016) https://arxiv.org/pdf/1603.06937.pdf

[2] Tompson, J.J., Jain, A., LeCun, Y., Bregler, C.: Joint training of a convolutional network and a graphical model for human pose estimation. In: Advances in Neural Information Processing Systems. (2014) 1799?1807

[3] Long, J., Shelhamer, E., Darrell, T.: Fully convolutional networks for semantic segmentation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2015) 3431?3440

[4] Farabet, C., Couprie, C., Najman, L., LeCun, Y.: Learning hierarchical features for scene labeling. Pattern Analysis and Machine Intelligence, IEEE Transactions on 35(8) (2013) 1915?1929

[5] Andriluka, M., Pishchulin, L., Gehler, P., Schiele, B.: 2d human pose estimation: New benchmark and state of the art analysis. In: Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on, IEEE (2014) 3686?3693

[6] Wei, S.E., Ramakrishna, V., Kanade, T., Sheikh, Y.: Convolutional pose machines. Computer Vision and Pattern Recognition (CVPR), 2016 IEEE Conference on (2016)

[7] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. Computer Vision and Pattern Recognition, 2016. CVPR 2016. IEEE Conference on (2015)

[8] Tieleman, T., Hinton, G.: Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning (2012)

[9] Kingma, D., Ba, J.: Adam: A Method for Stochastic Optimization, 2015. ICLR Conference on (2015)

[10] Kathuria, A: Intro to optimization in deep learning: Momentum, RMSProp and Adam, 2018. [Online]. Available: https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop- adam/. [Accessed: 19- Nov- 2018]

[11] Tompson, J., Goroshin, R., Jain, A., LeCun, Y., Bregler, C.: Efficient object localization using convolutional networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2015) 648?656

[12] Pishchulin, L., Insafutdinov, E., Tang, S., Andres, B., Andriluka, M., Gehler, P., Schiele, B.: Deepcut: Joint subset partition and labeling for multi person pose estimation. Computer Vision and Pattern Recognition (CVPR), 2016 IEEE Conference on (2015)

[13] . Carreira, J., Agrawal, P., Fragkiadaki, K., Malik, J.: Human pose estimation with iterative error feedback. Computer Vision and Pattern Recognition (CVPR), 2016 IEEE Conference on (2016)

[14] Hu, P., Ramanan, D.: Bottom-up and top-down reasoning with hierarchical rectified gaussians. In: Computer Vision and Pattern Recognition (CVPR), 2016 IEEE Conference on, IEEE (2016)