Automated rearfoot angle tracking on video images

Realization document

Bachelor's degree in Applied Computer Science
field of AI development

Matej Buršík

Academic year 2024-2025

Campus Geel, Kleinhoefstraat 4, BE-2440 Geel

Table of Contents

1 INTRODUCTION

The document serves as a realization document for the internship assignment titled *automated rearfoot angle tracking on video images*. The internship is being completed by Matej Buršík, a bachelor’s degree student in Applied Computer Science specializing in Artificial Intelligence development. The internship is conducted at Materialise Motion. It was supervised by Laurien Stroobants and the intern was mentored by Sam Van Rossom.

The outline of the internship assignment was defined in a document called project plan. It contains the internship goals, the technical approach to the assignment, and the timeline for its completion.

In summary, the internship is going to take thirteen weeks, starting 24th of February and ending 23rd of May. During these weeks, the assignment was to investigate and develop a proof-of-concept workflow to extract the rearfoot angle for each leg from a video or an image. The aim of this proof-of-concept workflow and potential application implementation would be to allow clinicians to easily make this measurement enhancing their ability to create custom shoe inserts for patients.

The purpose of the realization document is to cover everything that was achieved during the completion of the internship assignment and what was the outcome of the internship assignment.

The realization document consists of three important parts: an analysis, the results, and a conclusion. Analysis contains the general research and decisions made during completion of the assignment. This would include information about the tools, libraries, architecture, or methodologies used and the decision-making process behind the selection of the right tools. Results contain the description of the final product that is handed over to the company with an explanation of how it can be used. This part should also contain diagrams of workflows and design of applications. Lastly, there is a conclusion that contains the recapitulation of the most important information from the previous sections. It should also cover the evaluation which compares the results to the objectives stated in the project plan.

2 ANALYSIS

As you may have read in the project plan, there are two different approaches that were developed during the internship. One utilizing a pretrained model and one where the model was completely custom made. The analysis therefore is going to be divided into these four sections. Two sections talking about the models individually, one section mentioning some differences that were observed while working with them, and the last section discussing the test application.

All development was done in Python due to its ability to quickly prototype and develop solutions and having access to a wide range of very useful libraries.

All tests that are mentioned in this section can be found in the Result section.

2.1 Pretrained model research

A pretrained model is an artificial intelligence model that has already been trained on a large dataset before being used for a specific task. Instead of starting from scratch, developers use a pretrained model as a foundation and fine-tune it for their specific needs, which saves development time and computational resources. By using a pretrained model in the workflow, it allowed me to spend more time working on the data instead of working on developing the model.

2.1.1 Which model to use?

There were few models that could be used to complete the task of keypoint prediction, but not all suit the exact specifications that were required by Materialise Motion. In general, the most important metric in the end was that the solution could be commercially used, since the previous attempts used a model which was not. In order to choose a suitable model, a comparison table was created to evaluate the available options. The data for this table was compiled from documentation pages of the individual models and discussion forums attached to them.

Model	License	Developer	Key Features & Use case	Ease of Use	Community Support
YOLOv11	AGPL-3.0	Ultralytics	Object detection, instance segmentation, pose estimation, used for real-time applications	High	Very Strong
Detectron2	Apache-2.0	Facebook AI Research	State-of-the-art detection and segmentation algorithms, used for research and production applications	Moderate	Strong
MMPose	Apache-2.0	OpenMMLab	Pose estimation toolbox, used for human pose analysis	Moderate	Growing

DeepLabCut	LGPL-3.0	Developed by Mathis et al.	Markerless pose estimation for animals and humans, used for behavioral studies, animal tracking	Moderate	Moderate
OpenPose	Non-commercial, research only	CMU Perceptual Computing Lab	Real-time multi-person keypoint detection for body, used for academic research	Moderate	Moderate
AlphaPose	Non-commercial, research only	MVIG-SJTU	Accurate multi-person poses estimation and tracking, used for academic research	Moderate	Moderate

Just by looking at the License column, most of the models got ruled out, leaving only Detectron2 and MMPose. To decide between these two, the remaining columns were evaluated in which Detectron2 won. Detecron2 won due to it having relatively strong community support compared to MMPose, it being used in production level applications, and when both of them were tried out, there were much more issues setting up the MMPose model than there were setting up the Detectron2 model. (Ultralytics, 2025) (Ultralytics, n.d.) (Facebookresearch, n.d.) (Open-Mmlab, n.d.) (*DeepLabCut — the Mathis Lab of Adaptive Intelligence*, n.d.) (DeepLabCut, n.d.) (Cmu-Perceptual-Computing-Lab, n.d.) (Mvig-Sjtu, n.d.)

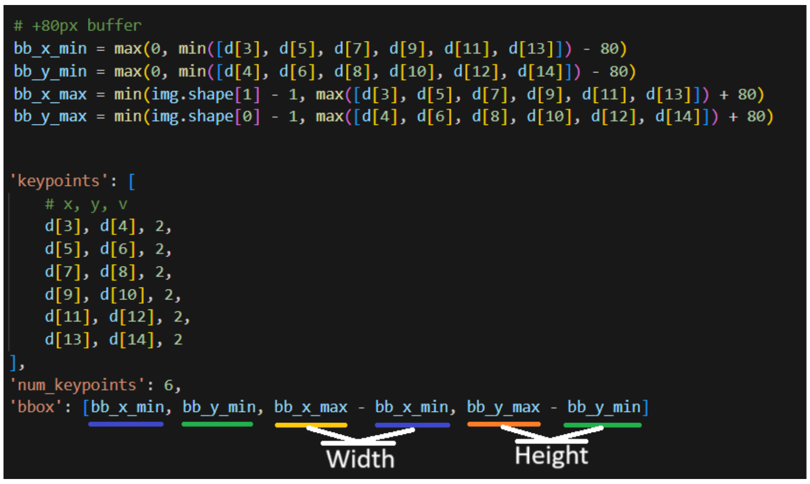
2.1.2 Dataset

During the introduction to the assignment, my mentor provided me with a labeled dataset which originally was from a previous attempt of this assignment. It contained sub-sets with images and a CSV containing the keypoint labels. There were around 750 fully labeled images present in that dataset. Although it was a relatively small dataset, it was already a good starting point.

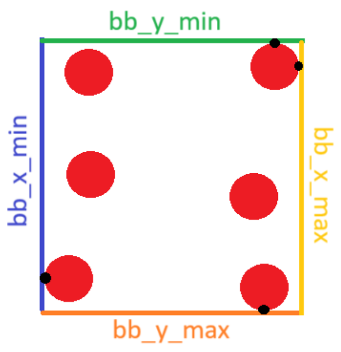
The dataset needed to be restructured into the COCO format for compatibility with Detectron2 because COCO is a widely used standard for object detection, segmentation, and keypoint detection. Detectron2 dataloader is optimized to work with COCO style datasets, which provide annotations in a structured JSON format rather than CSV files.

By converting the dataset, it ensures that Detectron2 can efficiently interpret the annotations, including image metadata, bounding boxes, and keypoints. The COCO format also facilitates training and evaluation by standardizing the dataset structure. (Lin et al., 2014)

The original dataset did not include bounding box information. To generate this data, the keypoint values were used to calculate the bounding boxes that encapsulated all keypoints for each instance.



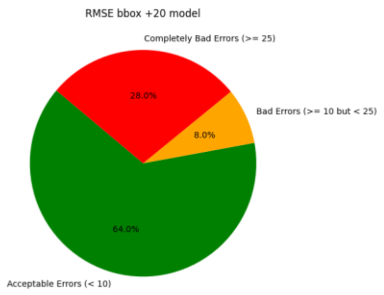
```
# +80px buffer
bb_x_min = max(0, min([d[3], d[5], d[7], d[9], d[11], d[13]]) - 80)
bb_y_min = max(0, min([d[4], d[6], d[8], d[10], d[12], d[14]]) - 80)
bb_x_max = min(img.shape[1] - 1, max([d[3], d[5], d[7], d[9], d[11], d[13]]) + 80)
bb_y_max = min(img.shape[0] - 1, max([d[4], d[6], d[8], d[10], d[12], d[14]]) + 80)


'keypoints': [
    # x, y, v
    d[3], d[4], 2,
    d[5], d[6], 2,
    d[7], d[8], 2,
    d[9], d[10], 2,
    d[11], d[12], 2,
    d[13], d[14], 2
],
'num_keypoints': 6,
'bbox': [bb_x_min, bb_y_min, bb_x_max - bb_x_min, bb_y_max - bb_y_min]
```

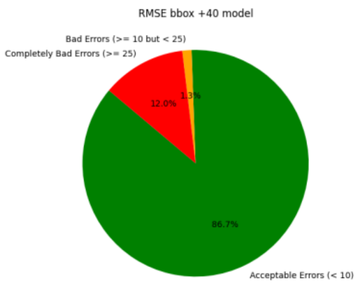


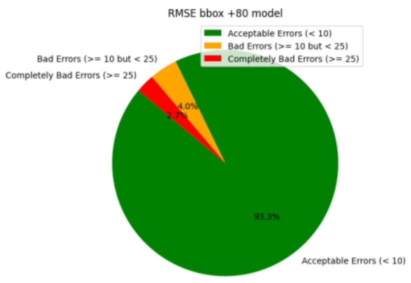
In the image, the bounding box is calculated by first identifying the minimum and maximum x and y coordinates from the keypoints. To ensure the bounding box captures not only the keypoints but also a margin of surrounding context, 80 pixels are added as a buffer. Specifically, 80 pixels are subtracted from the minimum x and y values (ensuring they don't go below 0), and 80 pixels are added to the maximum x and y values (ensuring they don't exceed the image boundaries). The resulting coordinates define the top-left corner and the width and height of the bounding box and effectively expanding it around the keypoints to include more visual context from the image.

I experimented with three different values for generating the bounding boxes. These are values that are added as buffers around the keypoints, letting the data contain more information about the surrounding of the keypoints. The chosen values were +20 pixels, +40 pixels, and +80 pixels.

These are the results of three models where each was trained on a different dataset with different bounding box sizes. The model then completed predictions of a test set on which an RMSE was calculated. The errors are then categorized into Acceptable which was below 10-pixel error, Bad which was between 10-pixel error and 25-pixel error, and Completely Bad which was above 25-pixel error. Very noticeably, the +80 model outperformed all the other categories.

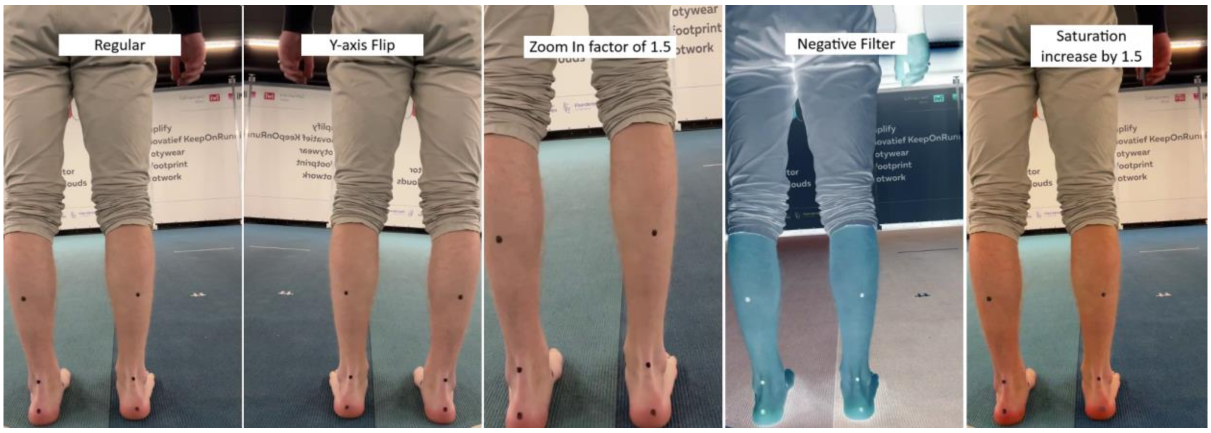






2.1.2.1 Data augmentation

In order to get better generalization of the model and prevent overfitting, it is common to use different forms of data augmentation. There were a few augmentation methods that were utilized to improve the dataset. Y-axis flip to introduce variability in orientation. Zoom in by a factor to introduce variability in scale of the targeted objects in the image. Color augmentation to reduce the racial imbalance in the dataset improving inclusivity. This was attempted but not tested if it actually helped with those issues. The approach was done by trying different color shifting effects like negative filter or simply increasing saturation.



2.2 Custom model research

Custom model is an artificial intelligence model that is designed and implemented from scratch. These models are tailored to meet the specific needs of a given use case. Designing a custom model involves developing a network architecture, choosing training loss functions, and configuring other hyperparameters in a manner that reflects constraints and performance requirements.

Training custom models typically requires more computational resources and a deeper understanding of the underlying learning dynamics. However, they offer unparalleled flexibility and can yield superior results in specialized domains where pretrained models may not perform adequately. Furthermore, custom architecture is instrumental in exploratory research, where novel structures or mechanisms are utilized.

Since the model was custom made, the dataset format could be defined freely. For the sake of simplicity and compatibility, the format that was used aligned with the format of Detectron2.

2.2.1 Which framework to use?

There are several frameworks which are frequently used to create custom models. In this case, the selection process allowed for greater flexibility, as the only requirement was that the framework be customizable and appropriate for the intended use case. The data for this table was compiled from documentation pages of the individual frameworks and discussion forums attached to them.

Library	**Ease of Use**	**Customizability**	**Use case**	**Community Support**
PyTorch	High	Very High	Research, experimentation, custom models	Strong and active
TensorFlow	Moderate	Very High	Production deployment, multi-platform use	Very strong and widespread
Keras	Very High	Moderate	Quick prototyping for TensorFlow, high-level APIs	Very strong

FastAI	Very High	Moderate	rapid prototyping, high-level APIs	Growing
JAX	Moderate	Very High	High-performance computing, scalability	Smaller but slowly growing
MXNet	Moderate	Moderate	Multi-language support, enterprise-level apps	Active in specific domains

Customizability is one of the higher priorities on the list of requirements and use cases. Based on that PyTorch, TensorFlow, and JAX are good contenders. Out of those, the decision landed on Pytorch due to its use case being more research oriented and it has a strong community support with good documentation. (*PyTorch*, n.d.) (TensorFlow, n.d.) (Team, n.d.) (*Welcome to Fastai – Fastai*, n.d.) (*JAX: High Performance Array Computing — JAX Documentation*, n.d.) (*Apache MXNet*, n.d.)

2.2.2 Network architectures

Designing an effective neural network architecture for keypoint detection involves a balance between feature extraction, model depth, and generalization ability. This chapter presents the development and comparative evaluation of multiple architectures ranging from simple convolutional layers to more advanced technics like residual blocks. The experiments were carried out on 256×256 and 512×512 images with varying hyperparameters, optimization strategies, and training schedules. All the test results of models presented in this chapter were trained and tested on the same dataset. They were also evaluated by the same set of Loss functions: Mean Squared Error (MSE), Root Mean Squared Error (RMSE), L1 loss, Smooth L1 loss, and Intersection over Union (IoU) loss.

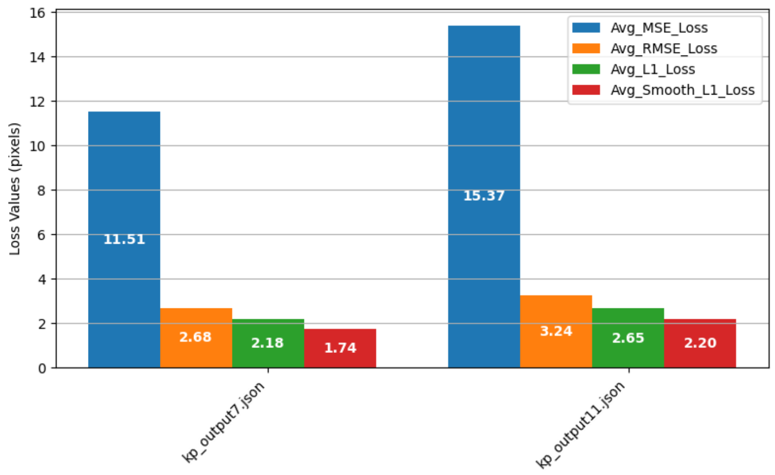
MSE and RMSE measure the average squared differences between predicted and actual keypoint locations, with RMSE providing a more interpretable metric by taking the square root of the MSE. L1 loss evaluates the absolute differences between predictions and ground truth, making it more robust to outliers compared to MSE. Smooth L1 loss introduces a transition between L1 and L2 loss by reducing sensitivity to small errors while penalizing larger discrepancies similarly to MSE. Finally, IoU loss assesses the overlap between predicted and ground truth bounding boxes, making it particularly useful for localization tasks where spatial accuracy is crucial. The use of these loss functions enabled a more objective comparison of the individual features.

When discussing the results, specific models will be referred to as for example keypoint_model_test0.pth and the data will show kp_output0.json. This structure follows all the tests performed on the custom model.

2.2.2.1 Baseline Convolutional Architecture

The initial architecture served as a baseline model featuring a simple stack of convolutional layers with increasing depth. Each convolution block was followed by Batch Normalization and ReLU activation, coupled with MaxPooling to downsample spatial dimensions. This setup was effective in learning basic spatial features.

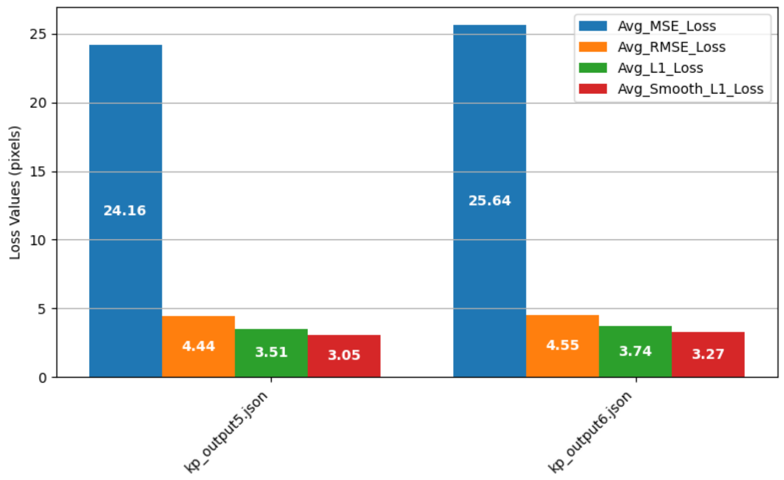
However, despite its simplicity and ease of training, this model plateaued early in terms of accuracy and struggled with capturing intricate spatial hierarchies necessary for precise keypoint predictions but still was able to outperform some of the more complex architectures. Good examples of this are tests done on keypoint_model_test7.pth and keypoint_model_test11.pth.



2.2.2.2 Feature extraction

Feature extraction is a technique used in machine learning and deep learning to utilize pretrained models like ResNet and MobileNet as feature extractors by processing images and identifying patterns. By extracting the features from the feature extractors allows the model to create a compact and informative representation of an image that reduces computational complexity while retaining crucial information.

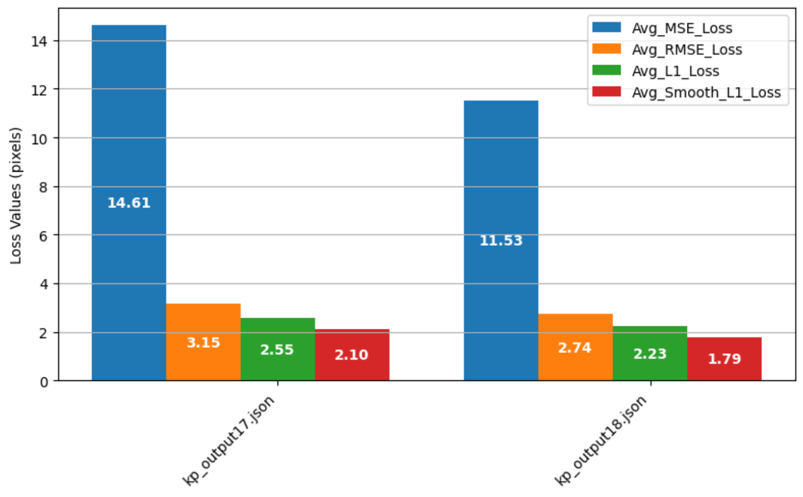
ResNet and MobileNet were tested as feature extractors in keypoint_model_test5.pth and keypoint_model_test6.pth. Contrary to expectations, both yielded underwhelming results, likely due to mismatches between the pretrained feature representations and the spatial granularity needed for keypoint regression.



2.2.2.3 Bottleneck layers

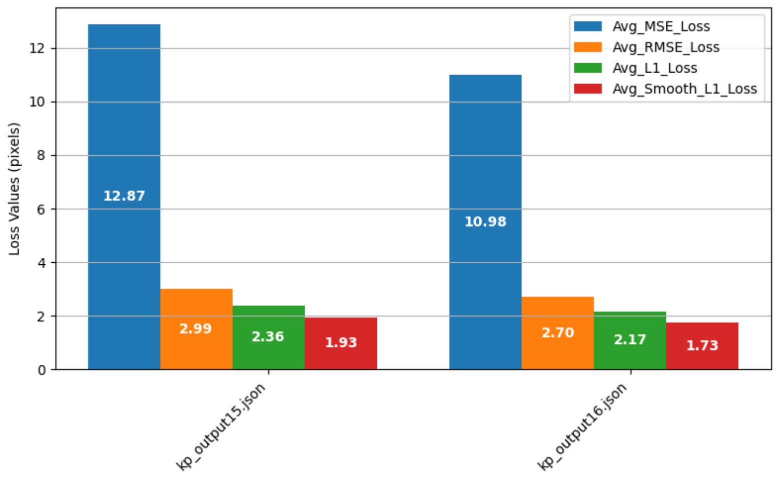
Inspired by ResNet’s bottleneck design, **traditional bottleneck layers** were incorporated into the architecture. A bottleneck layer was designed to reduce the number of parameters while preserving important information. It compresses the representation by reducing the number of features. Usually consists of a 1x1 convolution, followed by a larger convolution 3x3 and another 1x1 convolution. This structure helps reduce computation and increase efficiency while maintaining model accuracy

This approach, tested in keypoint_model_test17.pth, helped reduce parameter count and improve feature compactness. The results showed solid performance, marginally behind more complex variant. The more complex variant being keypoint_model_test18.pth which utilized both traditional bottleneck block and residual blocks.



Later after finding a **non-traditional bottleneck block** design, it was also implemented. The difference was that the 1×1 convolutions were applied only after 3×3 convolution, rather than sandwiching them.

This yielded excellent results, particularly in keypoint_model_test15.pth and keypoint_model_test16.pth. This strategy proved highly effective in enhancing feature diversity and maintaining resolution while adding minimal overhead. The difference being keypoint_model_test16.pth utilized residual blocks while keypoint_model_test15.pth did not.



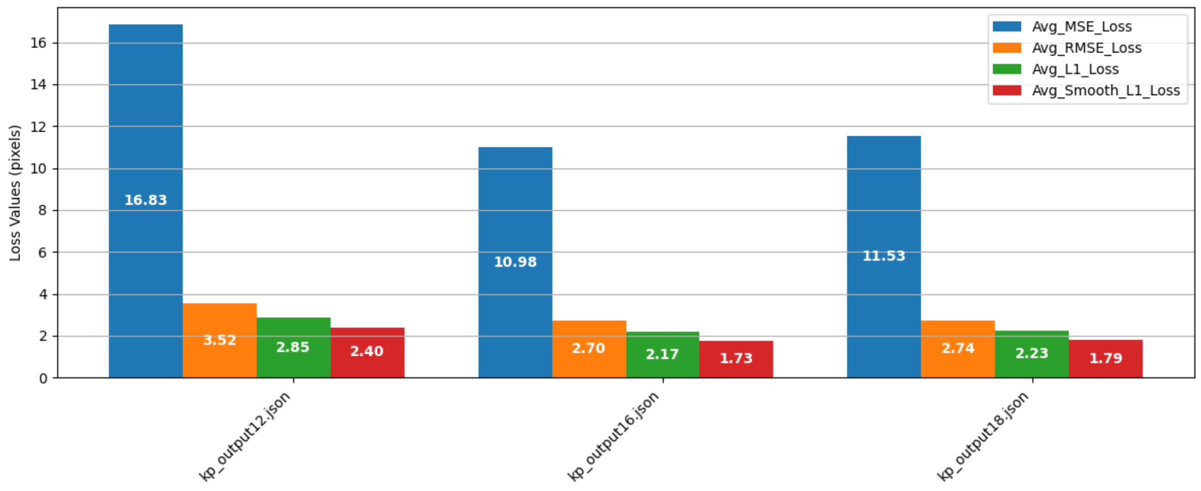
2.2.2.4 Residual Blocks

A residual block is a neural network building component designed to preserve information and ease gradient flow by introducing skip connections. These skip connections counteract issues with vanishing gradient in deeper neural networks.

Structure of a residual block: Conv3x3 -> BN -> ReLU -> Conv3x3 -> BN + SkipConnection -> ReLU

Architecture using residual block: RB -> MaxPool

Residual blocks led to consistent performance improvements, especially when combined with bottleneck designs. Notably, the architecture in keypoint_model_test12.pth (Residual only) and keypoint_model_test16.pth (Residual + Non-traditional Bottleneck) outperformed those without residuals. The final architectural iteration combined residual blocks with both traditional and non-traditional bottlenecks, tested in keypoint_model_test18.pth. While these models were slightly more complex, they achieved some of the best results in terms of accuracy and stability. Despite a slight increase in computational cost, the keypoint_model_test16.pth approach offered a robust balance of feature use, spatial detail retention, and gradient flow.



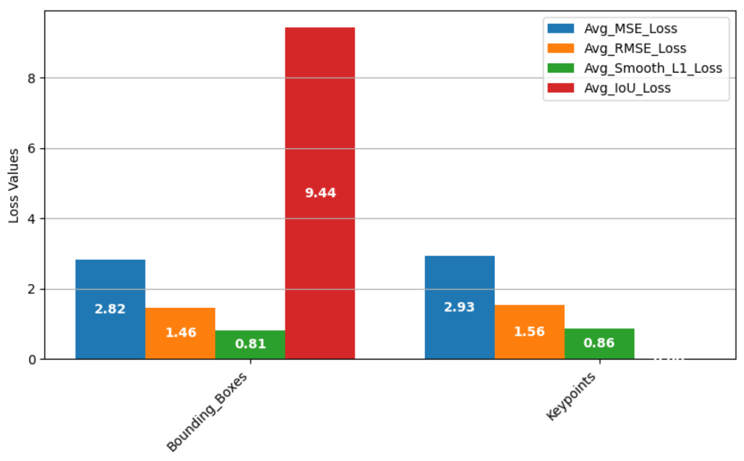
2.2.2.5 Multi-headed models

Compared to previous architectures, multi-headed architecture focuses more on the last fully connected decision head. To be exact it is a network design with multiple output heads, each serving a distinct function, branching off a shared backbone. This approach allows the model to perform multiple tasks simultaneously while leveraging shared features learned from the backbone. In this case, it utilized the localization head responsible for determining the spatial position of objects within an image and producing bounding box coordinates, and a keypoint head focuses on identifying specific key points on the patient.

There are two main reasons that lead to the research and test of this architecture. First, combined training of different heads allows the model to learn richer and more generalizable features, often leading to better robustness in diverse scenarios. Second, the localization head and keypoint head benefit from shared feature representations, making them more accurate compared to standalone models trained separately.

I also tested a different training approach called multi-stage training. During this training, the heads learn separately at first and then together. For example, during the first stage, only the localization head was learning. Then, during the second stage, only keypoint head was learning. Lastly, during the third stage, they trained together.

This architecture combined with this training technique resulted in the best custom model (multihead_model_test10).

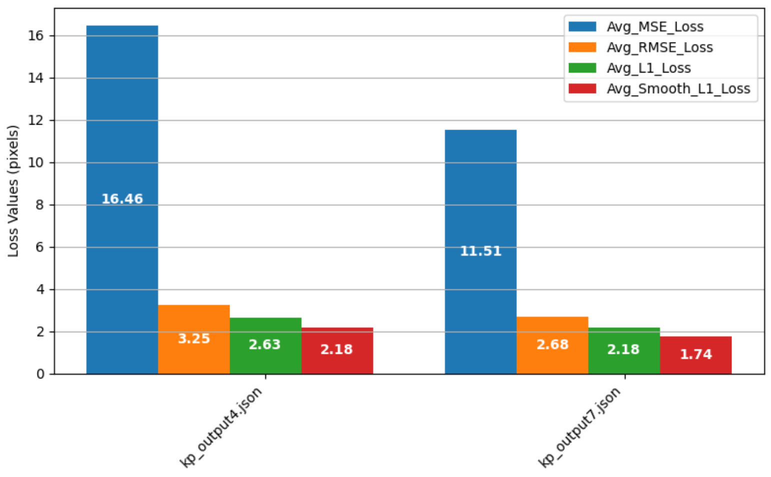


2.2.3 Model Fine-tuning

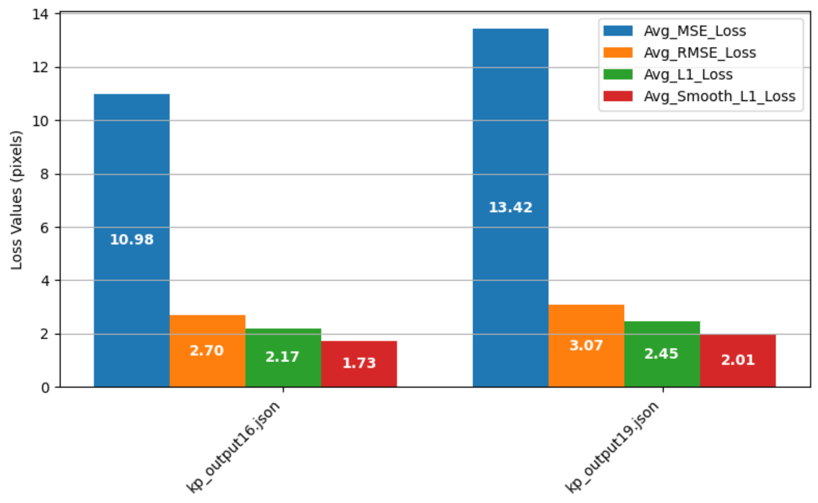
Hyperparameter tuning is the process of optimizing the parameters that control the behavior of a machine learning model. It is an important part of fine-tuning a model. Unlike model parameters, which are learned during training, hyperparameters are set before training begins and directly influence the model's performance.

Tuning involves systematically searching for the best combination of hyperparameters such as learning rate, batch size, weight decay, adaptive moment estimation, learning rate scheduling, and the loss function. These are some of the configurations that were tested and the results that were collected.

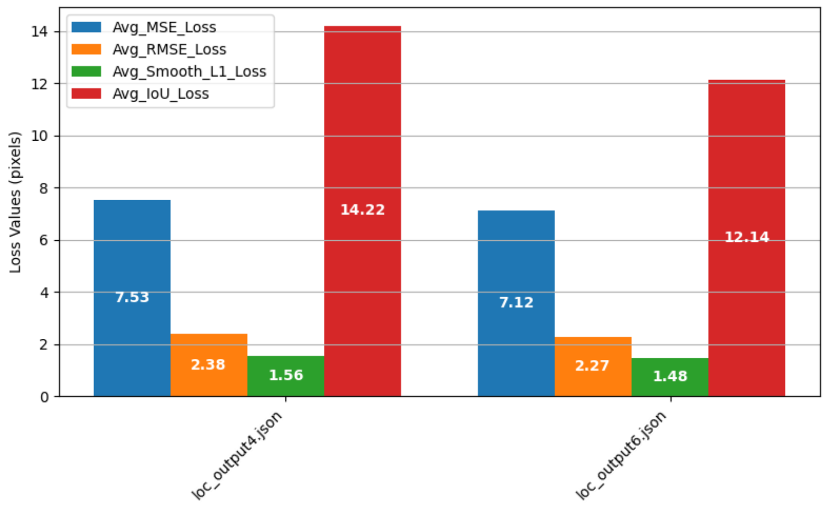
Learning rate scheduling is a technique used to optimize the training process of neural networks by dynamically adjusting the learning rate, preventing it from being too high or too low throughout training. Two common methods are StepLR and ReduceLROnPlateau. StepLR decreases the learning rate by a predefined factor at specific intervals, helping models gradually adapt to training complexities. On the other hand, ReduceLROnPlateau monitors validation loss and reduces the learning rate when performance improvement stagnates. When applied to keypoint_model_test4.pth, StepLR was used, leading to evaluation results such as average MSE of 16.46 and average RMSE of 3.25, indicating moderate error levels. In contrast, keypoint_model_test7.pth used ReduceLROnPlateau with a factor of 0.05 and patience of 4, meaning the learning rate was only reduced when no improvement was observed for 4 epochs. This approach stabilized training and resulted in improved evaluation metrics, with average MSE dropping to 11.50 and average RMSE to 2.68, demonstrating better performance.



Testing MSE and RMSE during training showed notable differences in model performance. MSE, used in keypoint_model_test16.pth, resulted in lower overall errors (average MSE: 10.98, average RMSE: 2.70), indicating stable training. RMSE, applied in keypoint_model_test19.pth, produced higher losses (average MSE: 13.42, average RMSE: 3.07), suggesting greater sensitivity to large deviations. The results highlight that MSE may be preferable for smoother training, while RMSE amplifies significant errors.



Comparing Smooth L1 and RMSE for localization tasks highlights their impact on model performance. Smooth L1, used in localization_model_test4.pth, balances L1 and L2 losses, reducing sensitivity to outliers. It achieved average MSE: 7.53, average RMSE: 2.38, and average smooth L1: 1.56, but showed a relatively higher average IoU: 14.22, which measures localization accuracy. On the other hand, RMSE was applied in localization_model_test6.pth, yielded average MSE: 7.12, average RMSE: 2.27, and average smooth L1: 1.48, indicating slightly lower overall errors. Notably, the average IoU dropped to 12.14, suggesting improved localization accuracy. These results imply that RMSE contributed to better IoU performance, whereas Smooth L1 maintained stability but at the cost of higher localization error.



Adaptive moment estimation, also known as Adam is an optimization algorithm used in machine learning and deep learning to update model parameters efficiently. It combines the advantages of two other methods: momentum (which helps with smooth updates) and adaptive learning rate (which adjusts learning rates adaptively). However, it applies weight decay by modifying the gradients directly, which can lead to suboptimal regularization. AdamW is variation that decouples weight decay from the optimization process, applying it directly to the weights instead of the gradients. This improves generalization, making AdamW better for training deep learning models, especially those prone to overfitting. It is also important to state what is weight decay. It is a regularization technique used in machine learning to prevent overfitting by discouraging overly large weights in a model. It works by adding a penalty to the loss function, which pushes the weights toward smaller values during training. A custom weight decay value was tested however, it had no significant impact on the results when compared to the default value used by the AdamW optimizer.

The last hyperparameter that was tested was the dropout rate even though dropout is not truly a hyperparameter but a network layer. Dropout layer is a regularization technique used to prevent overfitting in neural networks. It works by randomly dropping (setting to zero) a fraction of the input units during training. This forces the network to learn more robust features by preventing it from relying too heavily on specific neurons. Best results for using dropout depend on the depth of the architecture and amount of data. The results over several different tests pointed to a dropout of 0.1–0.25 being optimal in avoiding overfitting without impeding learning.

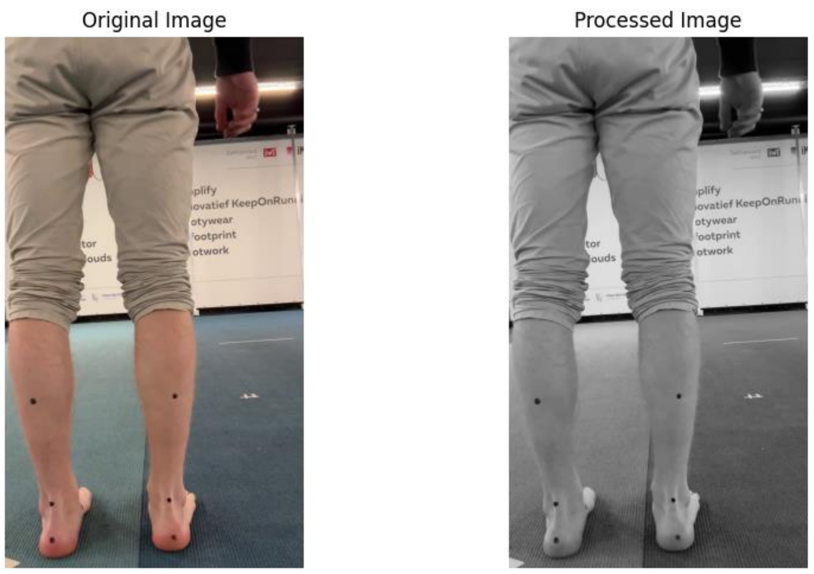
2.2.4 Data preprocessing

The focus of data preprocessing for the development workflow was to standardize the images that are going to be used for training and to enhance the images so that the model had the highest chance of having accurate predictions. This is usually achieved by different transformation or filter functions.

In the case of this internship, the code utilized for the subsections can be found in the `` `*Custom/data_preprocessing.ipynb*` `` subdirectory in the handover folder.

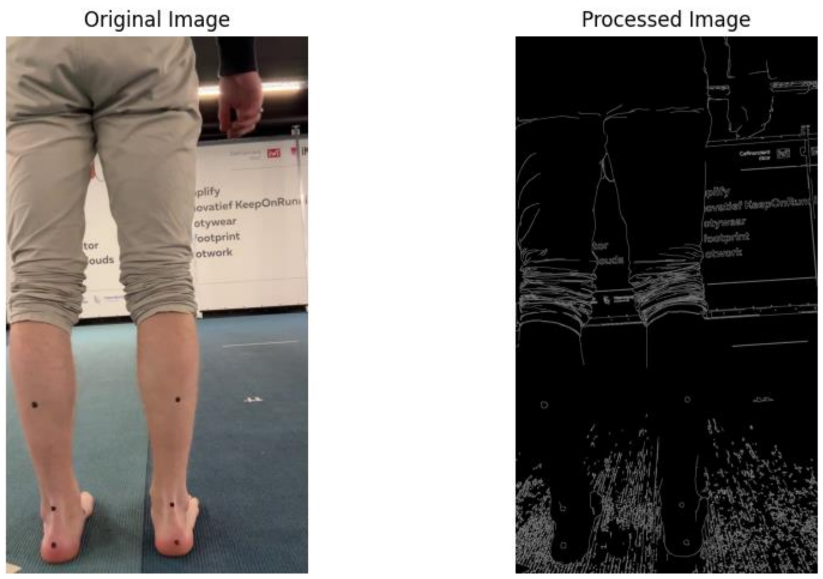
2.2.4.1 Gray scale

Gray scale reduces dimensionality by transforming color images into single channel instead of three (Red, Green, and Blue). This simplifies computations and allows models to focus on essential structural patterns rather than color variations. This also helps combat petential ratial bias that might be present in the dataset.

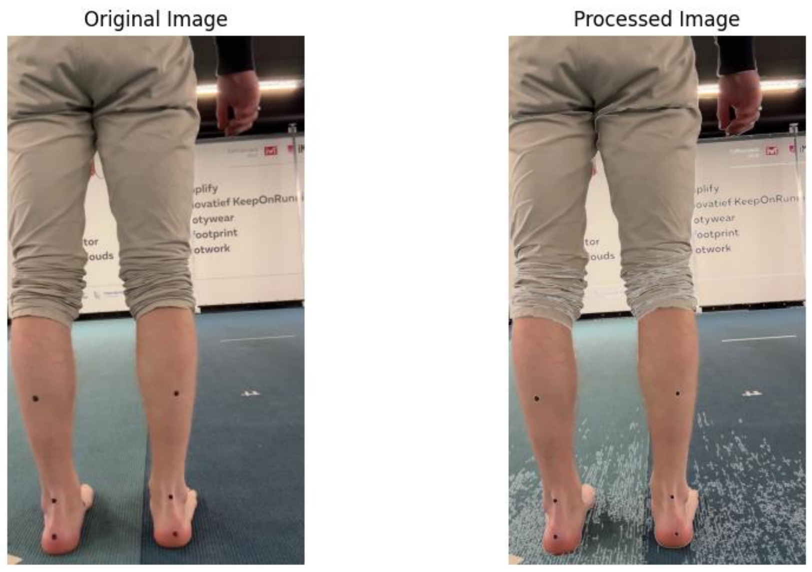


2.2.4.2 Edge enhancing

Edge enhancing highlights important edges and features by emphasizing transitions between pixel intensities. This is particularly useful for feature extraction tasks such as keypoint detection, making it easier for the model to identify key structures. In this case, the result is achieved by Canny edge detection which is a multi-stage algorith for detecting a wide range of edges in images. It is one of the most popular edge detection methods in computer vision systems due to its reliability. (Wikipedia contributors, 2025)



In some cases, having the edges was not enough because of motion blur. Motion blur cased some important features like the six markers to be hardly visible, making the edge detector does not catch them. A solution to this problem was to overlay the edges on top of the color image effectively highlighting important features but keeping the color information giving the model a high chance of perceiving the motion blurred markers.

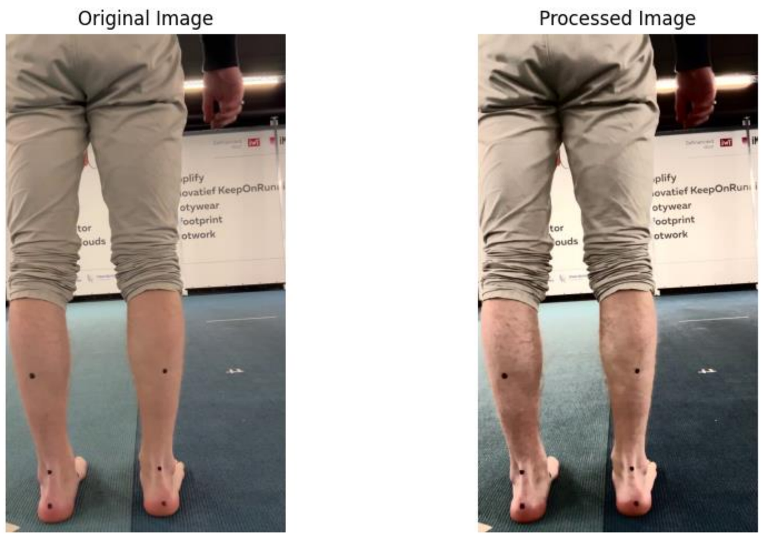


Based on the results from the tests keypoint_model_test20, keypoint_model_test21, and keypoint_model_test22, tests 21 and 22 which use the overlaying system outperform the test20 which only used edges. This was also proven by keypoint_model_test23 which used only edges while keypoint_model_test24 and keypoint_model_test25 used the overlaying system.

2.2.4.3 Contrast enhancing

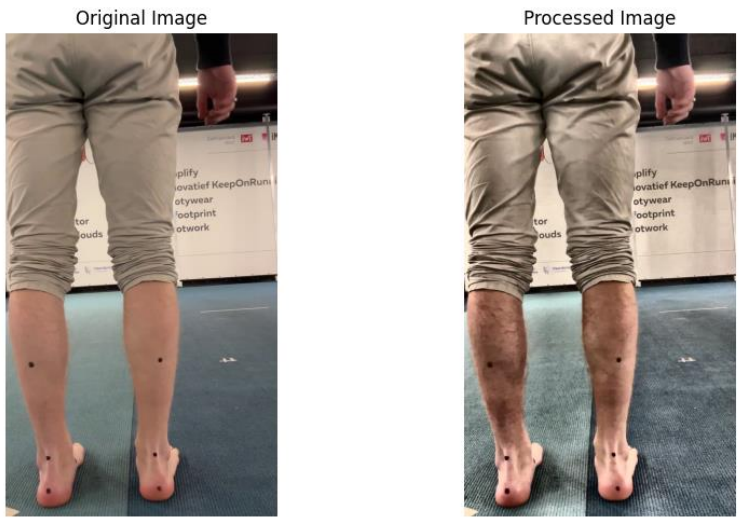
Adjusts pixel intensity distribution to improve visibility of features in low-contrast images. Methods like histogram equalization or CLAHE make details more distinguishable, leading to better feature extraction and recognition.

Result from Histogram color equalization.



At first the histogram equalization was not satisfactory, so CLAHE (Contrast Limited Adaptive Histogram Equalization) was applied which provides a very similar output but through different means. Histogram equalization enhances contrast by redistributing pixel intensities across the entire image. It works globally, meaning it applies the same transformation to all pixels, which can sometimes lead to over-enhancement or loss of detail in certain areas. CLAHE improves upon standard histogram equalization by applying localized contrast adjustments. It divides the image into small regions and equalizes each separately, preventing excessive contrast amplification. Additionally, it limits contrast enhancement to avoid amplifying noise.

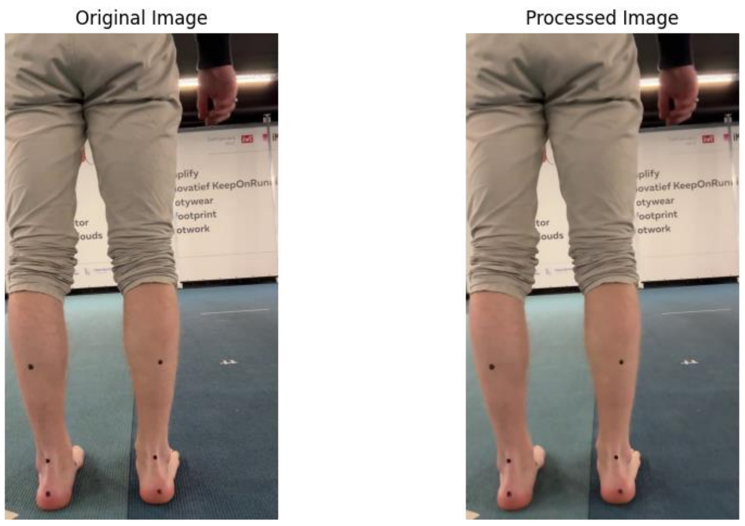
Result from CLAHE.



Even though CLAHE's approach is more sophisticated and in theory better the results show something different. Based on the results from keypoint_model_test21 (using CLAHE) and keypoint_model_test22 (using histogram equalization), the test22 outperformed the test21. The same result was repeated when it was tested with networks using residual blocks (keypoint_model_test24 using CLAHE and keypoint_model_test25 using histogram equalization).

2.2.4.4 Noise reduction

Noise reduction filters out unwanted artifacts and random variations in pixel intensities using techniques like Gaussian blur or bilateral filtering. This prevents the model from learning irrelevant noise, improving generalization and robustness.

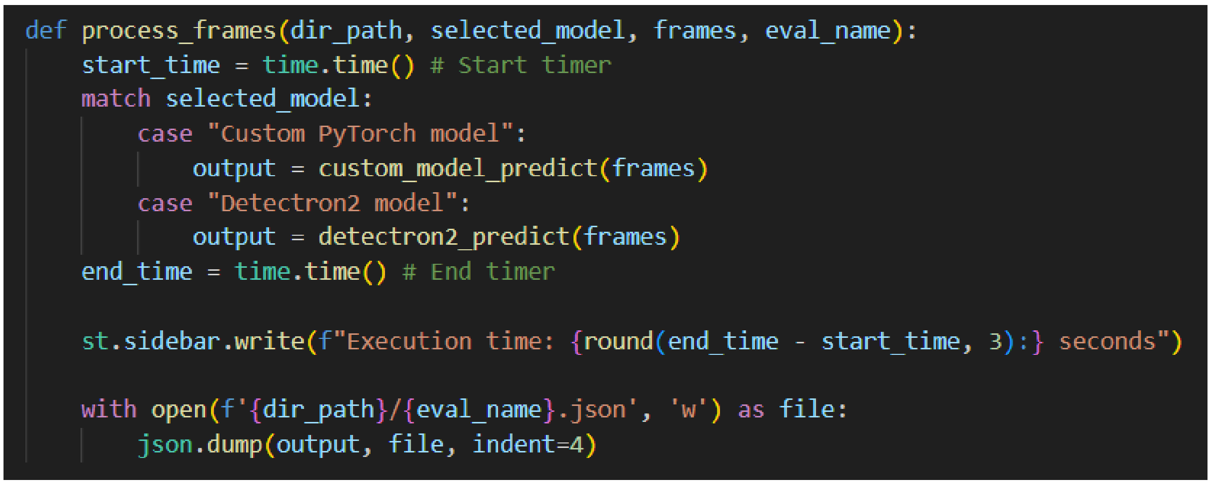


2.3 Difference between pretrained and custom model

Pretrained and custom models offer distinct advantages depending on the task and context. Pretrained models are developed on large, generic datasets and serve as a convenient starting point for new projects, allowing developers to save time and computational effort by fine-tuning these models for specific applications. This approach is particularly useful when working with limited data or when rapid development is a priority, as it enables more focus on data preparation rather than model architecture.

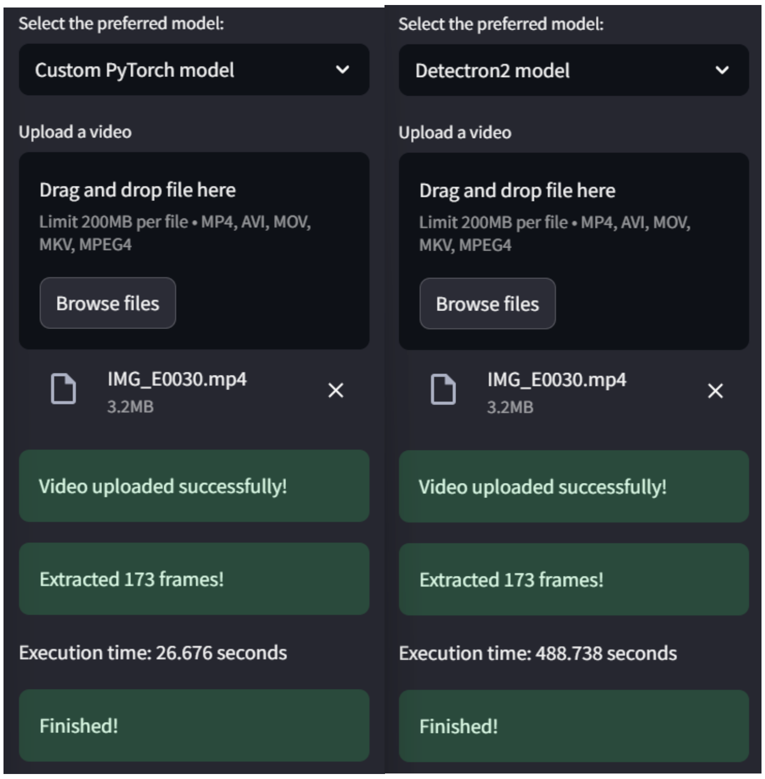
In contrast, custom models are built from the ground up, providing greater control over the general architecture of the model. While they demand more expertise, time, and resources, custom models can outperform pretrained ones in specialized or novel tasks, offering the flexibility required for research and unique applications where off-the-shelf solutions can fall short.

Since prediction speed was an important factor for this assignment, a speed test was performed, utilizing the test application to measure the time on both custom model and Detectron2. By implementing simple timer into the *process_frames* function, it was able to efficiently measure the execution time.



```python
def process_frames(dir_path, selected_model, frames, eval_name):
    start_time = time.time() # Start timer
    match selected_model:
        case "Custom PyTorch model":
            output = custom_model_predict(frames)
        case "Detectron2 model":
            output = detectron2_predict(frames)
    end_time = time.time() # End timer

    st.sidebar.write(f"Execution time: {round(end_time - start_time, 3):} seconds")

    with open(f'{dir_path}/{eval_name}.json', 'w') as file:
        json.dump(output, file, indent=4)
```

Model	**Number of Frames**	**Execution Time (Seconds)**
Custom PyTorch model Name: multihead_model_test8	173	26.676
Detectron2 model Name: output_5	173	488.738



Select the preferred model:
Custom PyTorch model

Upload a video
Drag and drop file here
Limit 200MB per file • MP4, AVI, MOV, MKV, MPEG4
Browse files
IMG_E0030.mp4
3.2MB

Video uploaded successfully!

Extracted 173 frames!

Execution time: 26.676 seconds

Finished!

Select the preferred model:
Detectron2 model

Upload a video
Drag and drop file here
Limit 200MB per file • MP4, AVI, MOV, MKV, MPEG4
Browse files
IMG_E0030.mp4
3.2MB

Video uploaded successfully!

Extracted 173 frames!

Execution time: 488.738 seconds

Finished!

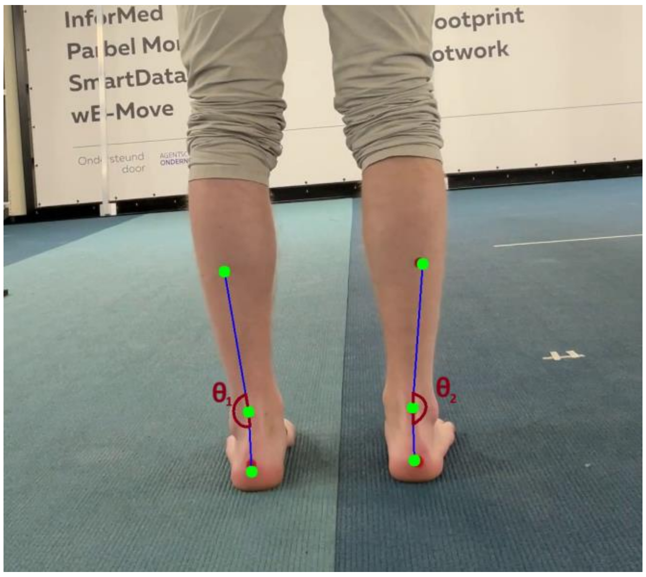
2.4 Test application research

One of the optional parts of this assignment was the development of a test application for the models. This section will cover some research that was done during its development. It is important to mention that the timeframe for the development was one week.

The first choice that had to be made was to pick a framework to create this application in. The chosen framework was Streamlit due to its ability to facilitate rapid development. Given the constrained development timeline, it was essential to select a framework that allowed for quick and easy development without requiring extensive configuration or complex coding. Streamlit allows me to focus on core functionality rather than intricate setup processes or styling. Additionally, its built-in support for interactive elements and seamless integration with Python-based data processing made it a suitable choice for making this AI application.

2.4.1 Angle calculation between keypoints

One of the most important parts of this project is the angle calculation between the predicted keypoints on each leg. Each leg contained three keypoints, creating two intersecting lines. The angle of interest was the angle created by the intersection of these two lines. Please refer to the image below for more clarity.



In the test application, the angle was calculated using the arctan2 function from the Numpy Python library. The arctan2 computes the inverse tangent using both X and Y values. It determines the angle in a full range from $-\pi$ to π, correctly identifying the quadrant. This is not the case with the arctan function which determines the the angle in range from $-\pi/2$ to $\pi/2$ and does not correctly identify the quadrant. (*NumPy – Understanding Tan(), Arctan(), and Arctan2() Functions (6 Examples) - Sling Academy*, n.d.)

To make the code more readable, a function was created that handles this calculation and here is the process to calculate the angle between two-line segments (so for one leg).

Since the two-line segments are defined by points $P_0(x_0, y_0)$, $P_1(x_1, y_1)$, and $P_2(x_2, y_2)$, calculate the change in their coordinates:

$$\Delta x_1 = x_1 - x_0, \quad \Delta y_1 = y_1 - y_0$$

$$\Delta x_2 = x_2 - x_1, \quad \Delta y_2 = y_2 - y_1$$

Then, the angle of the individual line segments is calculated using the arctan2 function:

$$\theta_1 = \arctan 2\,(\Delta y_1, \Delta x_1)$$

$$\theta_2 = \arctan 2\,(\Delta y_2, \Delta x_2)$$

Next, calculate the absolute difference between the two angles which gives the angle between the two-line segments:

$$\Delta\theta = |\theta_2 - \theta_1|$$

Since the angle between two lines is always between 0° and 180°, adjust $\Delta\theta$ if it exceeds π radians:

$$\text{If } \Delta\theta > \pi, \text{then } \Delta\theta = 2\pi - \Delta$$

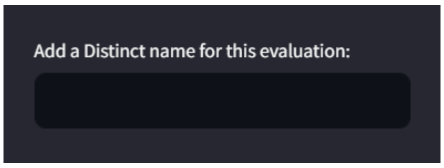
Finally, convert the result from radians to degrees:

$$\text{Angle (in degrees)} = \Delta\theta \cdot \left(\frac{180}{\pi}\right)$$

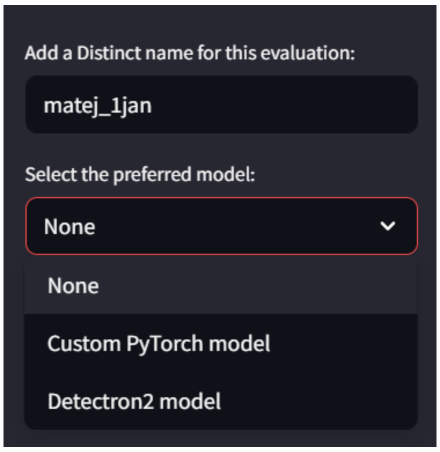
2.4.2 Application layout

This section is going to contain a description of the individual segments of the application's layout. There are two important portions of the layout: the sidebar, and the main container.

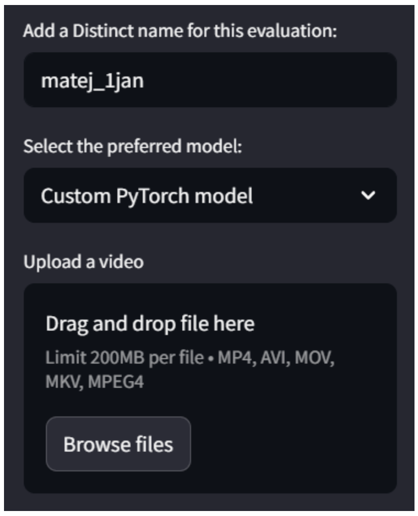
The sidebar focuses on processing of the video from a raw state, all the way to the predicted results and calculated angles. The sidebar starts only with a text input field in which the user inputs a distinct name for the evaluation they are going to perform. For example, the name of the patient plus the date it is performed.



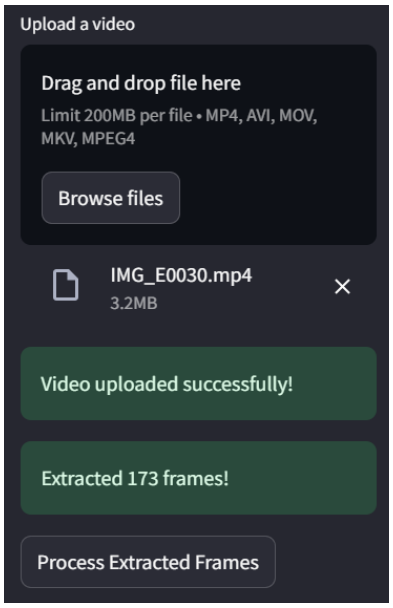
Next, a drop-down menu with a list of models pops up on the sidebar. The user can choose which model would they prefer to use. In this case they can select out of a Custom PyTorch model or a Detecron2 model.


Add a Distinct name for this evaluation:
matej_1jan
Select the preferred model:
None
None
Custom PyTorch model
Detectron2 model

When the user selects their preferred model, an upload zone for a video becomes available. Here the user uploads the video of the patient walking away from the camera with markers correctly placed on their legs. Please refer to the image in section 2.4.1 for more clarity.


Add a Distinct name for this evaluation:
matej_1jan
Select the preferred model:
Custom PyTorch model
Upload a video
Drag and drop file here
Limit 200MB per file • MP4, AVI, MOV, MKV, MPEG4
Browse files

After the user uploads a video, the video is saved and individual frames from the video are extracted. Then the user gets the option to process the extracted frames using the previously selected AI model.


Upload a video

Drag and drop file here

Limit 200MB per file • MP4, AVI, MOV, MKV, MPEG4

Browse files

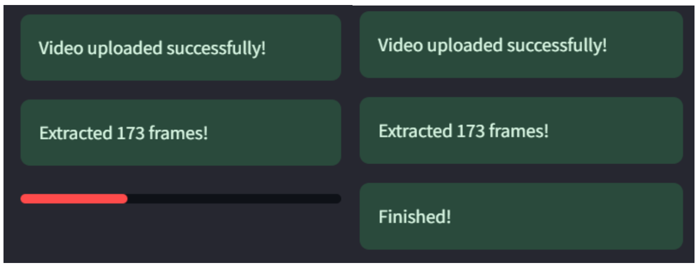
IMG_E0030.mp4
3.2MB

Video uploaded successfully!

Extracted 173 frames!

Process Extracted Frames

After pressing the button, a progress bar shows up to indicate how much time will it take to finish processing the images. When it is completed, a success message replaces the progress bar.


Video uploaded successfully!

Extracted 173 frames!

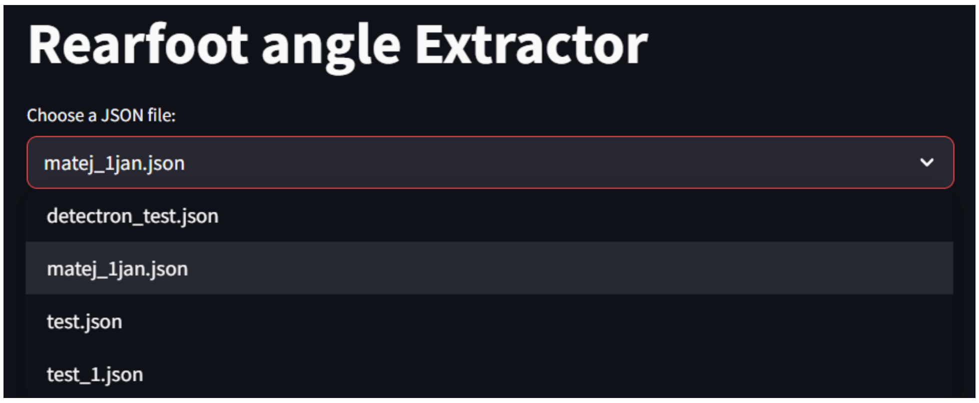
Video uploaded successfully!

Extracted 173 frames!

Finished!

This ends the functionally available in the sidebar.

The main container contains the visualization of results which were created by the processes in the sidebar. First, the user can choose which results do they want to see via a drop-down menu.


Rearfoot angle Extractor

Choose a JSON file:

matej_1jan.json

detectron_test.json

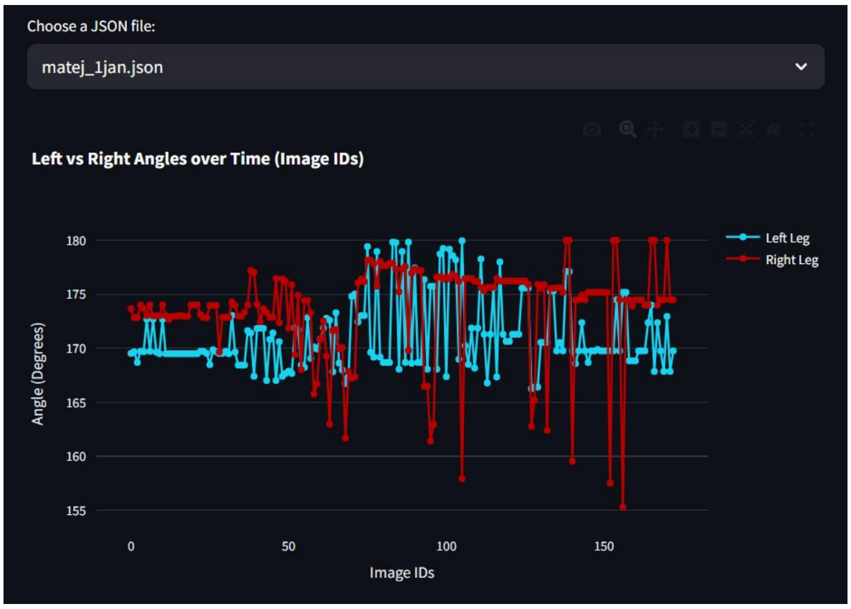
matej_1jan.json

test.json

test_1.json

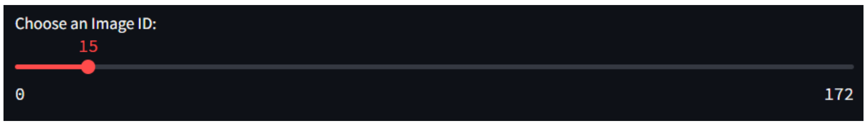
After selecting the result file which they want to visualize, a line graph of the angles from left and right legs over time. There are two traces on the graph, each being associated with a separate leg. The angle is displayed on the Y axis and time is on the X axis. Even though the X axis is not a concrete time definition, the image IDs give the user a chronological evaluation of the angles giving is more of a relative time definition.

The line graph was created using Plotly which allows it to be quite interactive compared to libraries like MatPlotLib. It allows hover to see data, zoom in and out, and apply filters without regenerating the graph.


Choose a JSON file:

matej_1jan.json

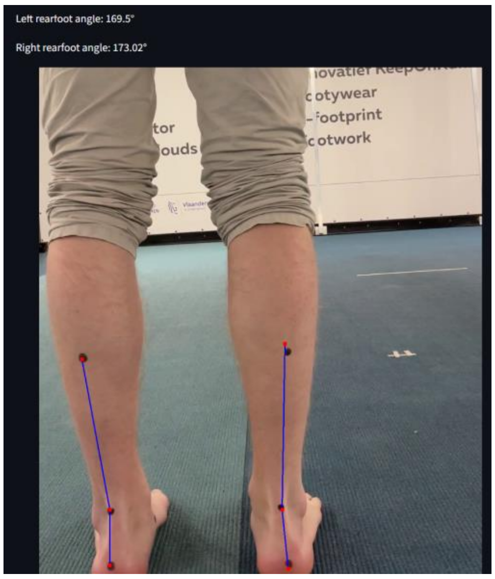
When the user has finished analyzing the line graph and has found which images do they want to see, they can select the ID of the image on the slider. In this case, it looks like the angle is quite stable for IDs in the range from 13 to 18, so ID 15 was chosen.


Choose an Image ID:

15

0 172

After selecting the image, a visualization of the prediction for that image is shown. This contains an image with an overlay of keypoints and the connections between the keypoints on individual legs. Above the image, there are written out the calculated rearfoot angles per leg.


Left rearfoot angle: 169.5°

Right rearfoot angle: 173.02°

3 Results

These tables contain the notes for all recorded tests and the results from both pretrained and custom models. At the end of each subsection, there is an explanation of which test resulted in the best model with the model's settings, architecture, and an explanation why it was the best. This model is also highlighted green in the result tables.

3.1 Detectron2 model

Test notes table for Detectron2 model

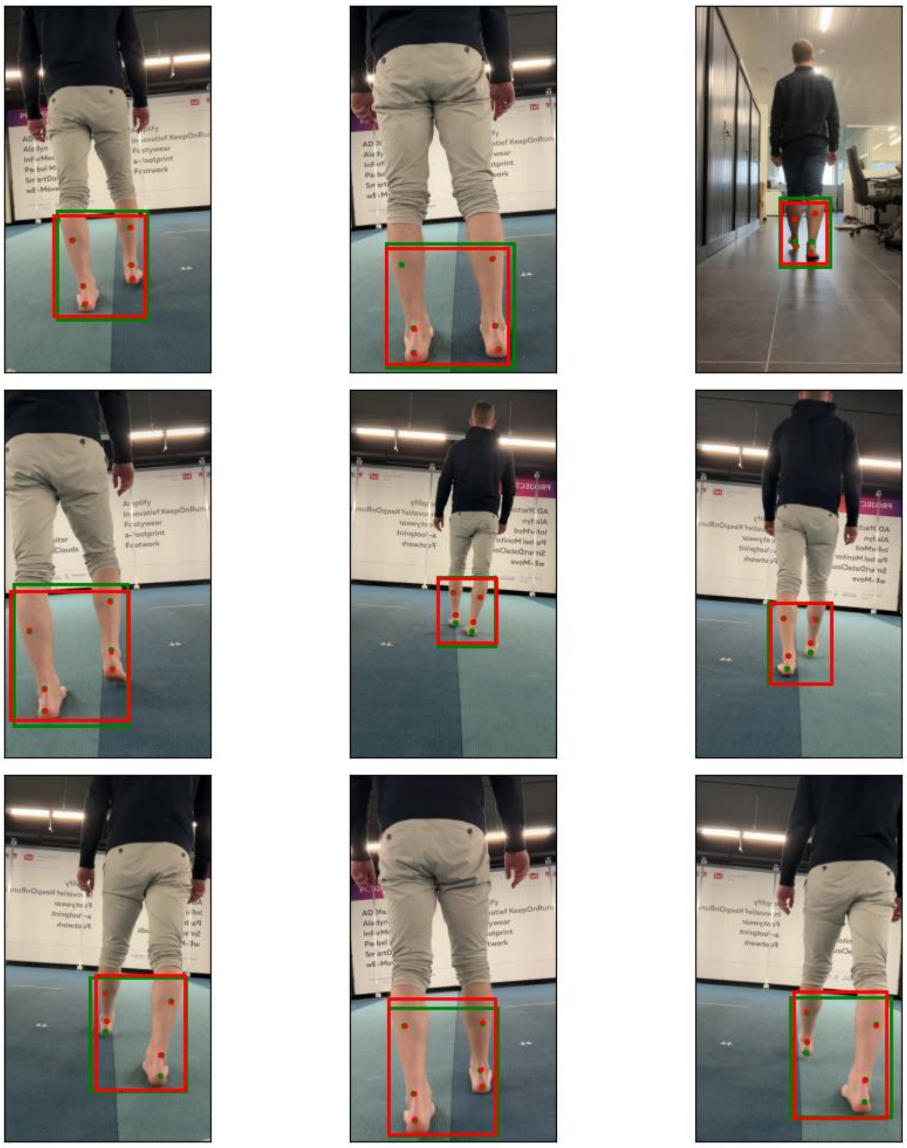
Model	Dataset and Augmentation	Configuration Notes	Results
output_1	Trained on manual_dataset, y-axis flip	Custom keypoint connections, model from COCO-Keypoints, 1000 iterations, evaluation every 100 iterations	Good evaluation, terrible video test, pose_test_1.mp4
output_2	Trained on expanded_20_bbox, 20px buffer, no augmentation	Same as output_1	Good evaluation, okay video, pose_test_2.mp4
output_3	Trained on expanded_40_bbox, 40px buffer, no augmentation	Same as output_1	Good evaluation, okay video, slight improvement, pose_test_3.mp4
output_4	Trained on expanded_100_bbox, 100px buffer, no augmentation	Same as output_1	Good evaluation, okay video, no real improvement, pose_test_4-1.mp4 and pose_test_4-2.mp4
output_5	Trained on expanded_80_bbox, 80px buffer, no augmentation	Same as output_1	The best model yet, pose_test_5-1.mp4 and pose_test_5-2.mp4
output_6	Trained on final, 80px buffer, y-axis flip, zoom for factors (1.4, 1.5, 1.6)	2000 iterations, evaluation every 250 iterations	Good evaluation, but bad video, logical error in zoom augmentation creating false data, pose_test_6-1.mp4 and pose_test_6-2.mp4
output_7	Trained on final_without_zoom, 80px buffer, y-axis flip	Same as output_1	Just slightly worse than output_5, pose_test_7-1.mp4 and pose_test_7-2.mp4

Test results table for Detectron2 model

Model	Iterations	BBox AP	BBox AP50	BBox AP75	Keypoints AP
output_1	1000	69.40	100.0	90.50	100.0
output_2	1000	74.51	99.97	90.50	99.97
output_3	1000	75.60	99.96	92.34	99.96
output_4	1000	73.05	100.0	87.74	100.0
output_5	1000	74.15	100.0	89.71	100.0
output_6	2000	83.74	100.0	98.96	100.0
output_7	1000	73.59	100.0	93.36	100.0

Model output_5

The evaluation of the output_5 model provides valuable insights into the model's performance across various metrics. The model was trained for 1000 iterations. The Bounding Box Average Precision (BBox AP), which measures detection accuracy based on Intersection over Union (IoU), was reported at 74.15, indicating a strong overall performance. The BBox AP50, which evaluated detections with an IoU threshold of 0.50, achieved a perfect score of 100.0, signifying the model's ability to reliably identify and localize objects at lower IoU thresholds. Similarly, the BBox AP75, which requires a stricter IoU threshold of 0.75, recorded a commendable score of 89.71, suggesting precise bounding box predictions. Additionally, the Keypoints AP, which measures the accuracy of keypoint detection, reached 100.0, demonstrating the model's exceptional capability in identifying and localizing individual keypoints with high precision. These metrics collectively indicate that the Detectron2 model performs reliably in detecting keypoints. The results suggest that the model was well-optimized and capable of handling complex detection tasks, making it a robust tool for various computer vision applications. This can furthermore be seen in the sanity check displayed below this text.



The Detectron2 keypoint_rcnn_R_50_FPN_3x architecture is a deep learning framework designed for keypoint detection and object recognition, based on Mask R-CNN. It utilizes ResNet-50 as its backbone for feature extraction, processing images through multiple residual layers to capture hierarchical patterns. To enhance detection across different scales, it incorporates a Feature Pyramid Network (FPN), which combines high-resolution spatial features with low-resolution semantic features.

The Region Proposal Network (RPN) generates candidate bounding boxes, which are refined through ROI Align to ensure accurate spatial alignment. The Keypoint Head then predicts heatmaps for individual keypoints, enabling precise localization. Additionally, the model includes bounding box and classification heads, ensuring simultaneous object detection and keypoint estimation.

Training follows a 3x schedule, meaning extended iterations for improved accuracy, using the COCO-Keypoints dataset with pretrained ImageNet weights to enhance feature extraction. This architecture is widely used for human pose estimation, gesture recognition, and sports analytics, offering robust performance in detecting and localizing keypoints with high precision. (Lad, 2024)

3.2 Custom keypoint model

Test notes table for custom keypoint model

Model	Image Size	Epochs	Adjustments	Result Comment
keypoint_model_test1	256x256	5	Simple convolutional network	No data output
keypoint_model_test2	256x256	20	-	No data output
keypoint_model_test3	256x256	25	BatchNorm2D, Dropout 0.5, AdamW	No data output
keypoint_model_test4	256x256	20	StepLR, no Dropout	Dropout was not needed yet
keypoint_model_test5 [FE]	256x256	20	ResNet feauture extraction	Underwhelming, no noticeable improvement
keypoint_model_test6 [FE]	256x256	20	MobileNet feauture extraction	Underwhelming, no noticeable improvement
keypoint_model_test7	256x256	20	No feature extraction, Replaced StepLR with ReduceLROnPlateau (factor=0.1, patience=3)	Good
keypoint_model_test8	256x256	20	ReduceLROnPlateau (factor=0.05, patience=4, threshold=0.001)	Quite good
keypoint_model_test9	256x256	20	ReduceLROnPlateau (factor=0.05, patience=5, threshold=0.001)	Suddenly extremely inconsistent predictions
keypoint_model_test10	256x256	30	ReduceLROnPlateau (factor=0.05, patience=4, threshold=0.001), Dropout 0.25	Needs more epochs or lower dropout
keypoint_model_test11	256x256	35	Dropout 0.1, Weight decay 0.0001	Quite good, could tweak

				epochs and dropout more
keypoint_model_test12 [RB]	256x256	35	Residual block architecture	Good
keypoint_model_test13	512x512	40	Dropout 0.2, No residual block	Quite bad
keypoint_model_test14 [RB]	512x512	40	Residual block architecture	Quite bad, slightly better
keypoint_model_test15	256x256	40	Non-traditional bottleneck layers, No residual block	Really good, best so far
keypoint_model_test16 [RB]	256x256	40	Non-traditional bottleneck layers, With residual block	Really good, better than nn without RB
keypoint_model_test17 [BN]	256x256	40	Traditional bottleneck layers, No residual block	Really good, slightly worse than non-traditional bottleneck
keypoint_model_test18 [BNRB]	256x256	50	Traditional bottleneck layers, With residual block	Really good, slightly worse than non-traditional bottleneck with RB
keypoint_model_test19 [RB]	256x256	40	Non-traditional bottleneck layers, With residual block, RMSE instead of MSE	Really good, slightly worse than MSE version
keypoint_model_test20	256x256	30	MSE, No RMSE, No residual block, Added edge-detection into preprocessing (only edges, not overlaid)	Quite good, has potential but trained on fewer epochs because the edges are simpler features
keypoint_model_test21	256x256	50	Updated preprocessing to include CLAHE and noise reduction	Quite good

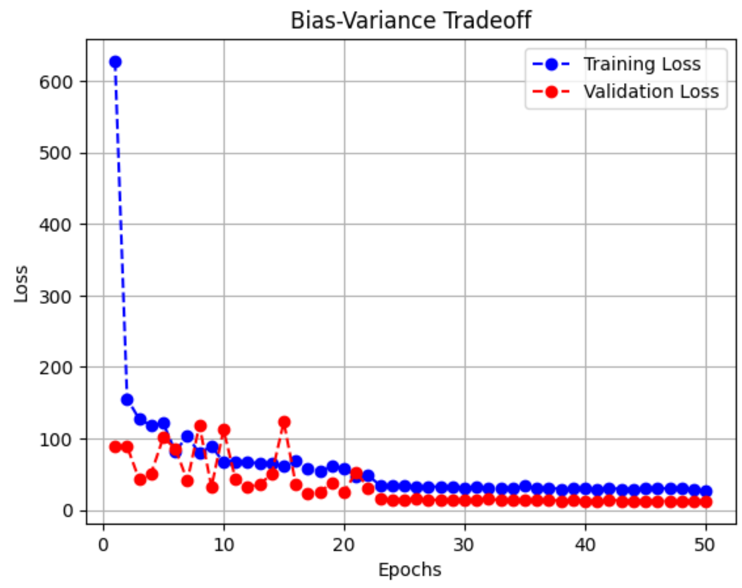
			before overlaying the edge detection	
keypoint_model_test22	256x256	50	Replaced CLAHE and noise reduction with custom color equalize and edge enhancing	Probably the best one yet, on par with test16
keypoint_model_test23 [RB]	256x256	50	nn with residual block, preprocessing has edge-detection (only edges, not overlaid)	Good, much better than the non-RB version
keypoint_model_test24 [RB]	256x256	50	nn with residual block, preprocessing has CLAGE and noise reduction	Much better than the non-RB version, on par with test16
keypoint_model_test25 [RB]	256x256	50	nn with residual block, preprocessing has custom color equalize and edge enhancing	Best keypoint so far, better than test16 or test24
keypoint_model_test26 [RB]	256x256	50	Preprocessing has only gray_scale()	Quite good, slightly worse than test16

Test results table for custom keypoint model

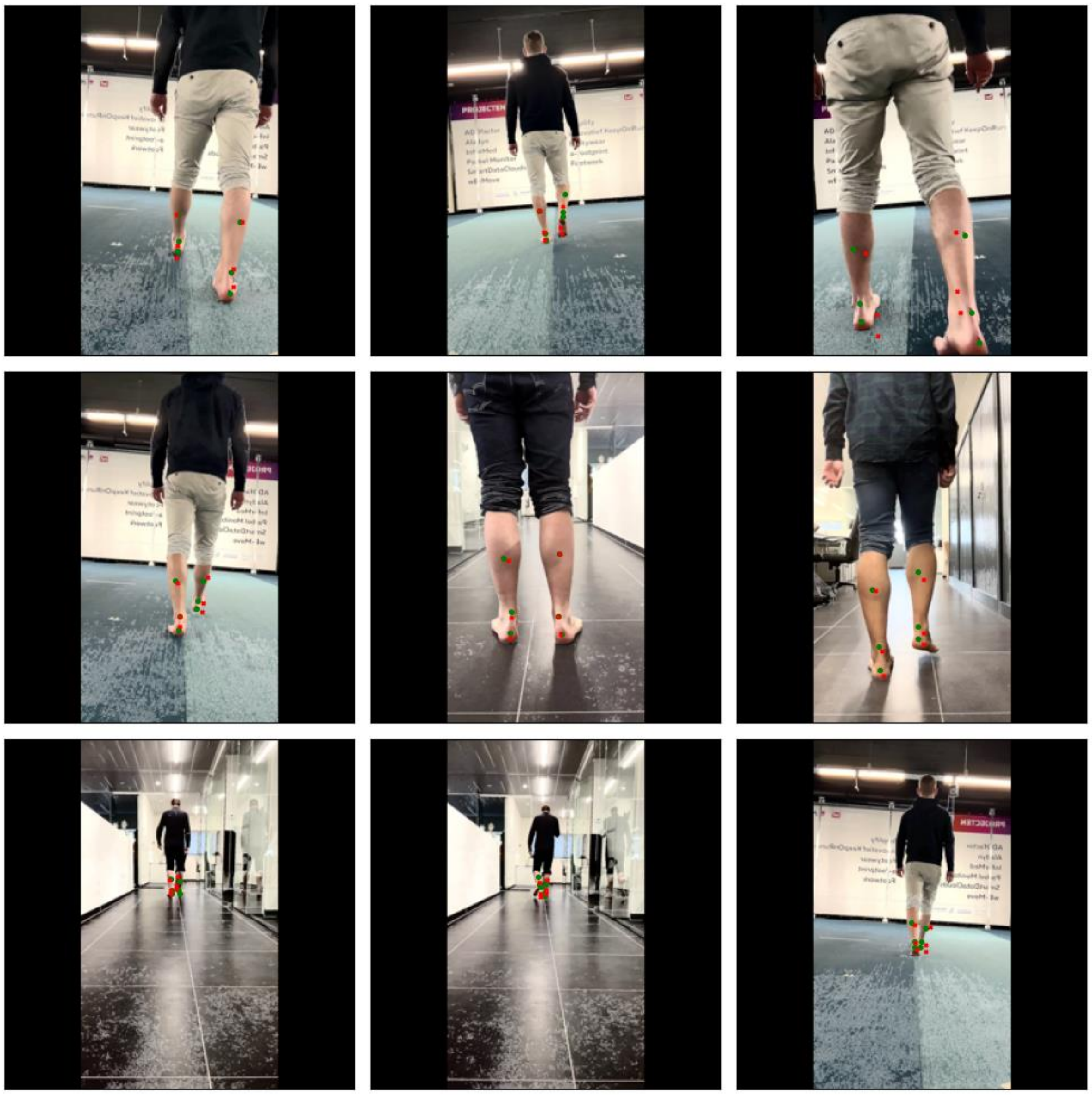
Model	Avg. MSE Loss	Avg. RMSE Loss	Avg. L1 Loss	Avg. Smooth L1 Loss
keypoint_model_test4	16.46464	3.24926	2.63320	2.18489
keypoint_model_test5	24.16326	4.43501	3.50648	3.04800
keypoint_model_test6	25.64307	4.54994	3.74286	3.27171
keypoint_model_test7	11.50533	2.67962	2.18081	1.74356
keypoint_model_test8	16.85154	3.12132	2.54005	2.09154
keypoint_model_test9	10.84975	2.69321	2.22959	1.78697
keypoint_model_test10	20.05236	3.95337	3.22602	2.76732
keypoint_model_test11	15.37092	3.24419	2.65357	2.20182

keypoint_model_test12	16.83220	3.51921	2.85476	2.40224
keypoint_model_test13	85.94661	7.64797	6.22190	5.74499
keypoint_model_test14	78.77646	7.14527	5.84012	5.36407
keypoint_model_test15	12.86808	2.98778	2.36341	1.92610
keypoint_model_test16	10.98398	2.69969	2.16839	1.72996
keypoint_model_test17	14.60561	3.15057	2.55280	2.10359
keypoint_model_test18	11.52689	2.73567	2.22778	1.78915
keypoint_model_test19	13.41606	3.06914	2.44727	2.00780
keypoint_model_test20	17.47051	3.71312	2.96553	2.51154
keypoint_model_test21	15.22028	3.24304	2.64864	2.19894
keypoint_model_test22	10.47890	2.68078	2.13970	1.69671
keypoint_model_test23	14.15748	3.34039	2.67428	2.22480
keypoint_model_test24	10.59707	2.67324	2.13681	1.70131
keypoint_model_test25	9.56481	2.62600	2.11804	1.68118
keypoint_model_test26	11.25277	2.84501	2.30825	1.86608

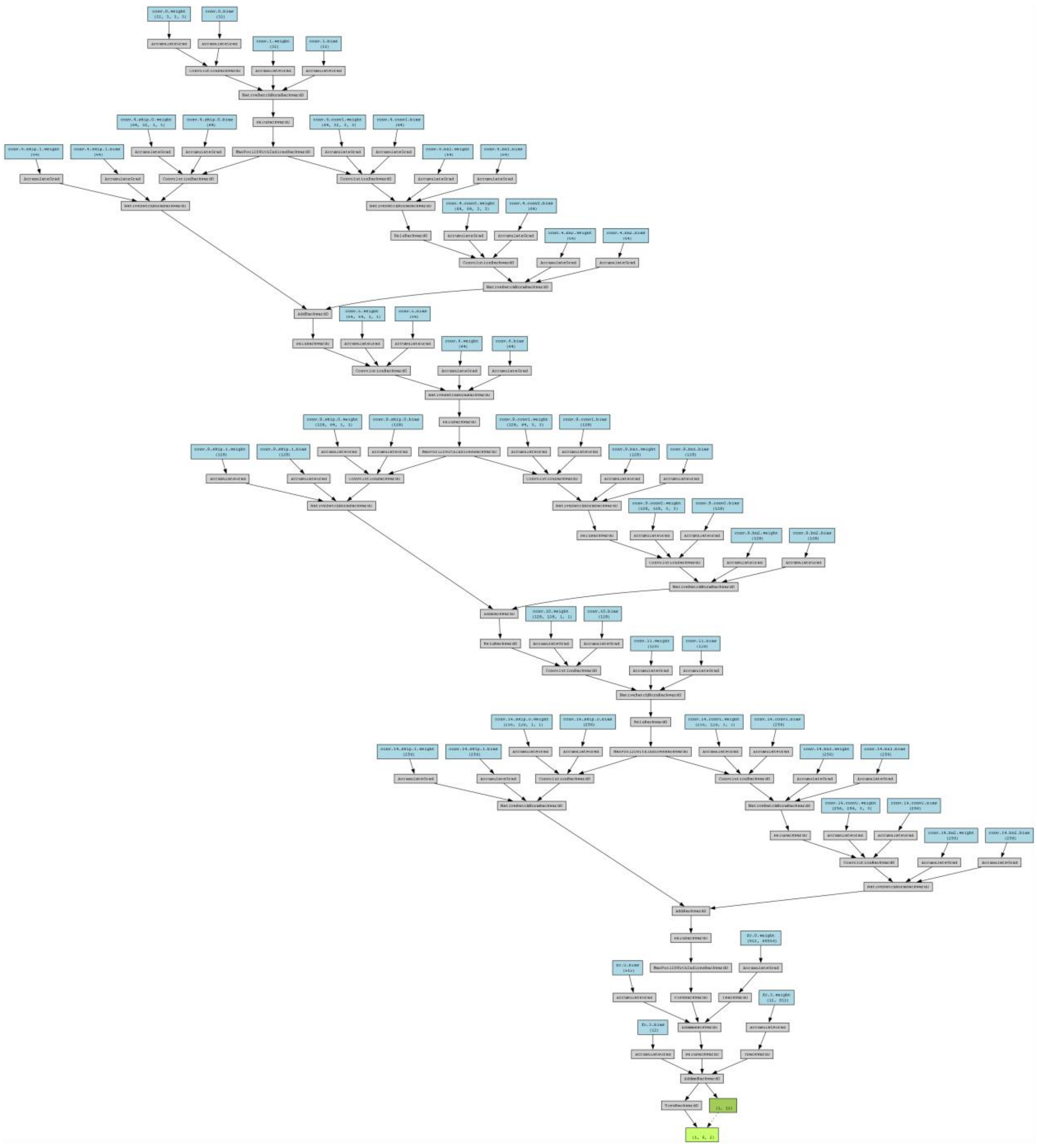
Model keypoint_model_test25



During the early training epochs, the model shows a significant reduction in bias, as training loss dropped from 627.40 to 121.55 by epoch 5, and validation loss also decreased to 101.46, indicating initial learning progress. However, between epochs 6-15, validation loss fluctuated wildly, moving between 86.07, 40.77, 118.70, 32.26, and 112.14, while training loss continued its downward trend. These large swings suggest some instability and potential overfitting, where the model might be overly sensitive to variations in training data. Fortunately, starting from epoch 16, validation loss started to stabilize between 25–15, while training loss continues decreasing at a slower rate, show improved generalization. At the end, training loss reached 28.61, and validation loss settled at 13.01, suggesting that the model has found a good balance between bias and variance, resulting in a well-generalized and stable training outcome.



It was obvious based on this sanity check that even though the model was heading in the correct direction there is plenty of room to improve. Most importantly, it was good to see that it was able to pick up on general patterns and do quite well on test dataset.



This architecture is designed for keypoint detection, leveraging residual blocks to enhance feature extraction and improve training efficiency. The residual block module facilitates smooth gradient flow through skip connections, ensuring stable optimization while mitigating vanishing gradients. The network comprises hierarchical convolution layers interspersed with max-pooling operations to systematically reduce spatial dimensions while preserving critical feature information. Additionally, 1x1 convolution bottleneck layers between residual blocks refine feature representation without excessive computational burden. The final fully connected output layer maps high-dimensional features to keypoint coordinates, generating precise (x, y) positions for six detected point. By combining residual learning principles with an optimized architecture, this model is well-suited for keypoint prediction and other vision tasks requiring detailed spatial predictions. This architecture shows a balance between accuracy and efficiency, demonstrating its potential for deployment on a handheld device.

3.3 Custom localization model

Test notes table for custom localization model

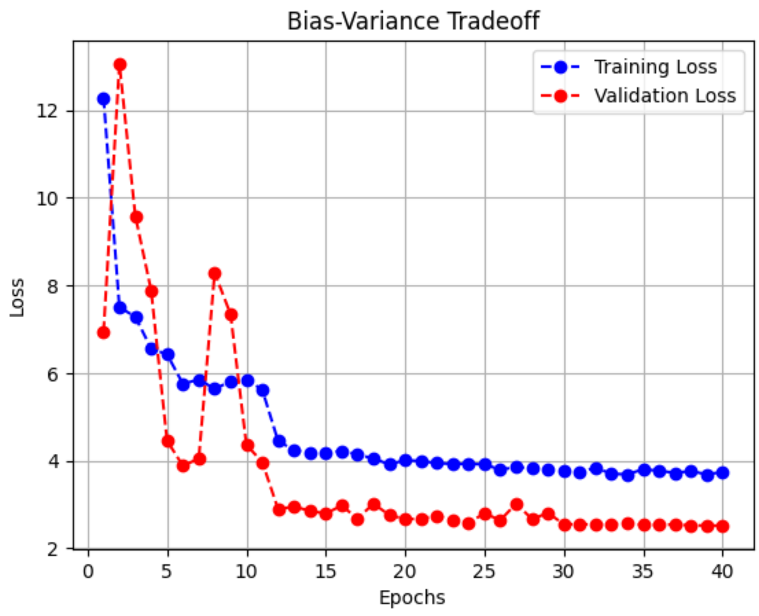
Model	Image Size	Epochs	Adjustments	Result Comment
localization_model_test1	256x256	40	Started from the keypoint test16 architecture but without residual blocks, StepLR(step_size=5, gamma=0.8), Training with SmoothL1Loss()	Really good, especially for the first localization model
localization_model_test2 [RB]	256x256	40	Added residual blocks	Good but though it would be better
localization_model_test3	256x256	40	Removed residual blocks, replaced StepLR with ReduceLROnPlateau	Good but though it would be better
localization_model_test4 [RB]	256x256	40	Added residual blocks	Really good, basically as good as test1
localization_model_test5 [BNRB]	256x256	40	Added traditional bottleneck layers	Good but worse than test4
localization_model_test6 [RB]	256x256	40	Removed traditional bottleneck layers, Replaced SmoothL1Loss with RMSE for training	Best localization model yet

Test results table for custom localization model

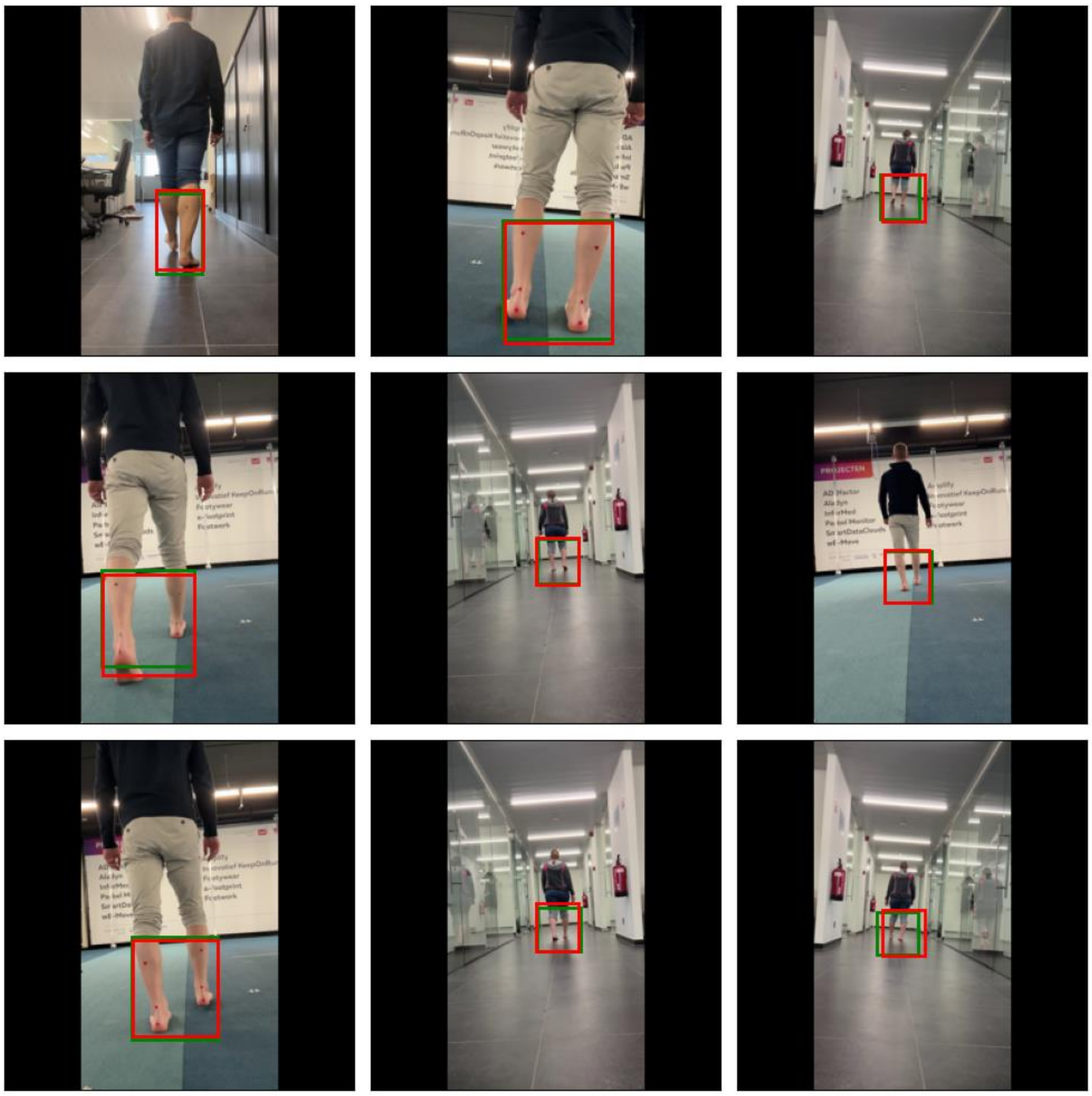
Model	Avg. MSE Loss	Avg. RMSE Loss	Avg. Smooth L1 Loss	Avg. IoU Loss
localization_model_test1	7.18410	2.34077	1.53004	13.82830
localization_model_test2	11.66381	3.03325	2.09618	21.70333
localization_model_test3	10.32671	2.71559	1.85093	14.61700
localization_model_test4	7.53199	2.38481	1.56312	14.21652

localization_model_test5	9.64061	2.69333	1.76863	17.09382
localization_model_test6	7.11733	2.26880	1.48339	12.13815

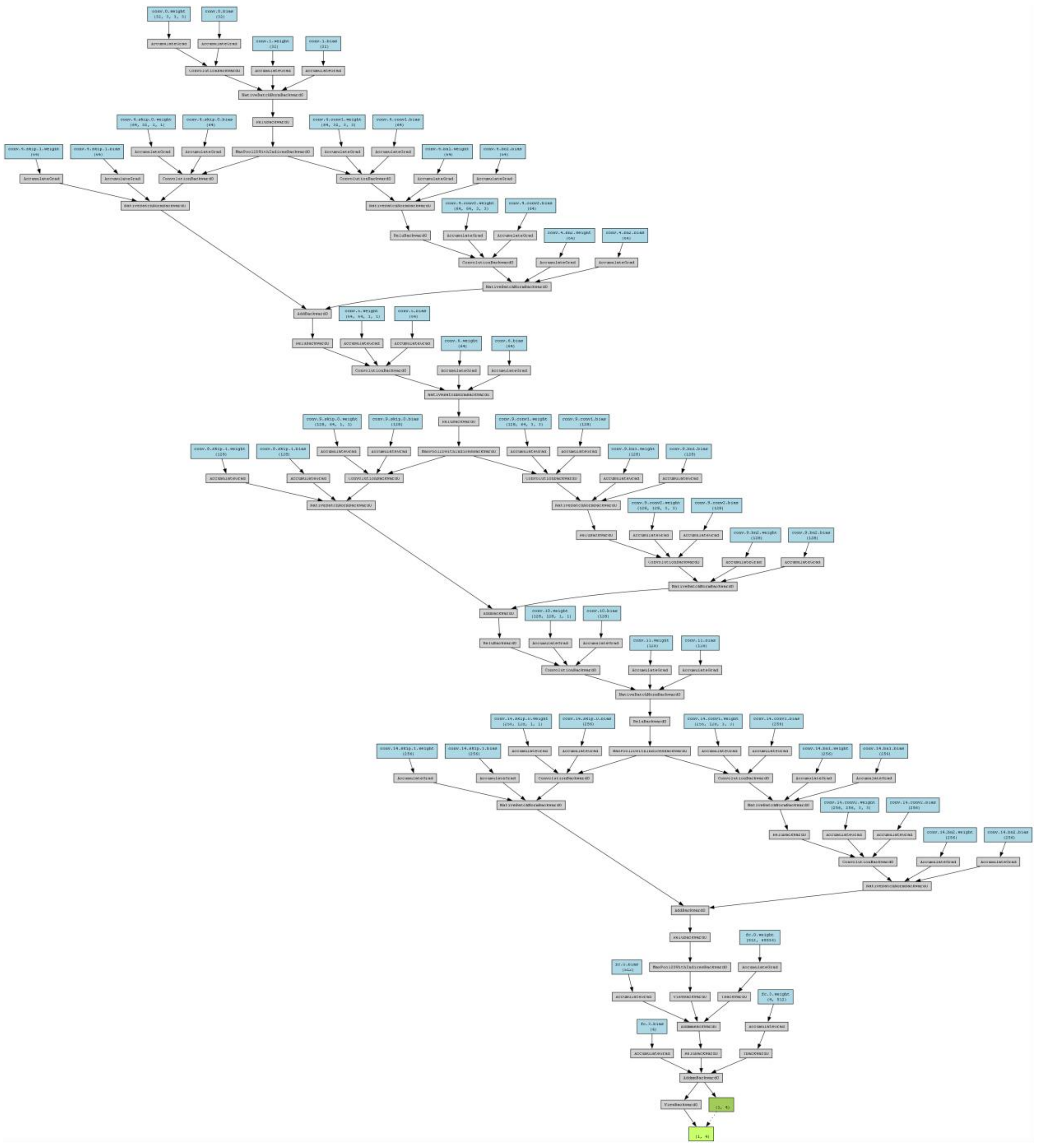
Model localization_model_test6



In the early epochs, the training process demonstrated a significant reduction in bias, with training loss dropped from 12.28 to 6.42 by epoch 5, and validation loss improved to 4.45, suggesting initial learning progress. However, in epoch 2, validation loss spiked to 13.06, which indicated instability, possibly caused by the model struggling to generalize. Between epochs 6-10, the validation loss fluctuated between 3.87, 4.04, 8.28, 7.34, and 4.35, while the training loss steadily declined. These fluctuations could suggest some variance-related overfitting, where the model is too sensitive to training data. Fortunately, from epoch 12, validation loss started to stabilize around 2.88–2.51, with training loss continuing to decline in a controlled manner. By epoch 35, training loss reached 3.73, and validation loss settled at 2.51, suggesting a well-balanced model that has learned effectively while maintaining a good generalization capability. If needed, regularization techniques like dropout might further refine the model's stability.



When compared to the sanity check of the custom keypoint model, it is noticeable that the localization model is much better at recognizing special features than the keypoint model. Based on that reason, these models were combined into a custom multi-headed model which is discussed in the next section.



This architecture is designed for localization tasks, employing residual blocks and 1x1 convolutional bottleneck layers in the same way as the keypoint model. The obvious difference is in the final stage which consists of a fully connected localization head, which processes high-level features to predict bounding box coordinates in the format [x_min, y_min, width, height].

3.4 Custom multi-headed model

Test notes table for custom multi-headed model

For test5 and onwards a multi-stage training was implemented during which the localization head trains separately first, then the keypoint head trains separately second, and then then train together. It is mentioned in the adjustments column and will be visible in the epochs column since each stage has its own number of epochs.

Model	Image Size	Epochs	Adjustments	Result Comment
multihead_model_test1	256x256	40	Architecture the same as localization test1, using ReduceLROnPlateau , Keypoint -> RMSE, BBox -> SmoothL1Loss, cropping uses ops.roi_align, forward feeds cropped output to keypoint head	Bbox head is okay, but keypoint head is terrible
multihead_model_test2	256x256	40	Removed cropping, forward feeds the original conv to the heads separately	Both bbox and keypoints where okay at best, but this is not the expected solution
multihead_model_test3	256x256	50	Cropping updated, now doing conv twice, once for full image before bbox, once for cropped image before keypoint	Bad, padding is confusing it, and it is incorrect, but it is technically better than previous cropping test
multihead_model_test4	256x256	50	Removed cropping, added masking to easily preserve the image shape	Bad, basically the same as test3
multihead_model_test5	256x256	BBox 35, KP 35, Joint 10	Implemented multi-stage training	Bad, basically the same as test3 and test4
multihead_model_test6	256x256	BBox 35, KP 35, Joint 10	No cropping or masking, combined backbone but completely separate prediction heads	Really good, slightly worse than the best separately trained models

multihead_model_test7	256x256	BBox 35, KP 35, Joint 30	Added residual blocks	Best model so far, even better than best keypoint models
multihead_model_test8	256x256	BBox 35, KP 35, Joint 30	Added custom color equalize and edge enhancing to preprocessing	Best model so far, better than test7
multihead_model_test9	256x256	BBox 35, KP 35, Joint 30	Replaced the training loss function from SmoothL1Loss to RMSE	Really good, equal to test7
multihead_model_test10	256x256	BBox 35, KP 35, Joint 30	Put back the SmoothL1Loss, Trained on dataset final_with_color_aug	Best model, unbelievably good, based on data collected while training it does not look overfitted

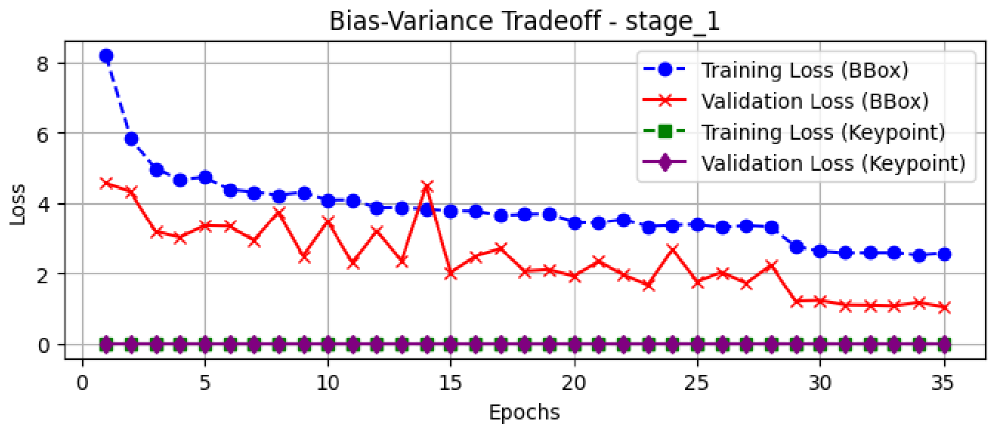
Test results table for custom multi-headed model

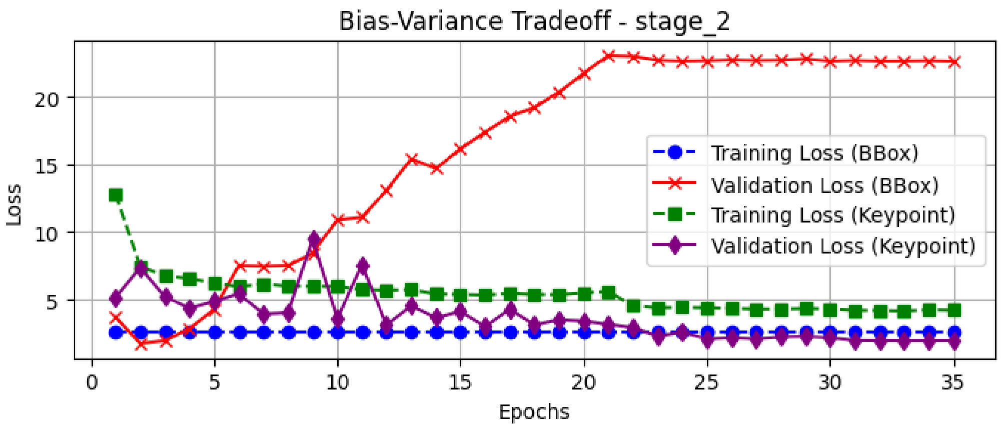
Since there were two predictions being made by this model there were also two different values in each cell. The first value was for bounding box and second value was for keypoints. Also there never was a calculation for average IoU loss for keypoints since there were six keypoints and the function used was made primarily for bounding boxes which have four keypoints.

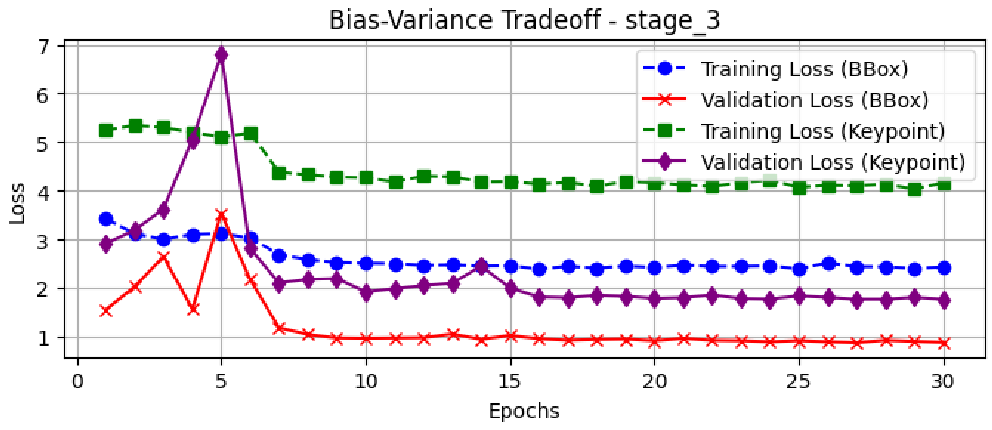
Model	Avg. MSE Loss	Avg. RMSE Loss	Avg. Smooth L1 Loss	Avg. IoU Loss
multihead_model_test1	9.70154 437.22588	2.66043 18.10628	1.80887 14.75663	13.99846 -
multihead_model_test2	11.26569 14.04511	2.84001 3.14663	1.95991 2.09533	14.52261 -
multihead_model_test3	337.48943 395.04973	17.64771 19.10901	15.18095 15.7175	62.79685 -

multihead_model_test4	261.63670 366.82000	15.90538 17.88944	12.54760 15.77884	77.70463 -
multihead_model_test5	240.22674 631.60123	15.12881 23.54829	11.98147 21.26195	75.07950 -
multihead_model_test6	10.15993 13.20910	2.77157 3.07703	1.86614 2.06505	16.45879 -
multihead_model_test7	6.55529 8.93353	2.25753 2.54773	1.47163 1.62858	13.59201 -
multihead_model_test8	5.48596 7.62177	1.98152 2.24935	1.24268 1.42338	11.28106 -
multihead_model_test9	5.95756 8.76575	1.97699 2.34503	1.25247 1.48542	10.83602 -
multihead_model_test10	2.82281 2.92729	1.46047 1.55537	0.81319 0.86477	9.44327 -

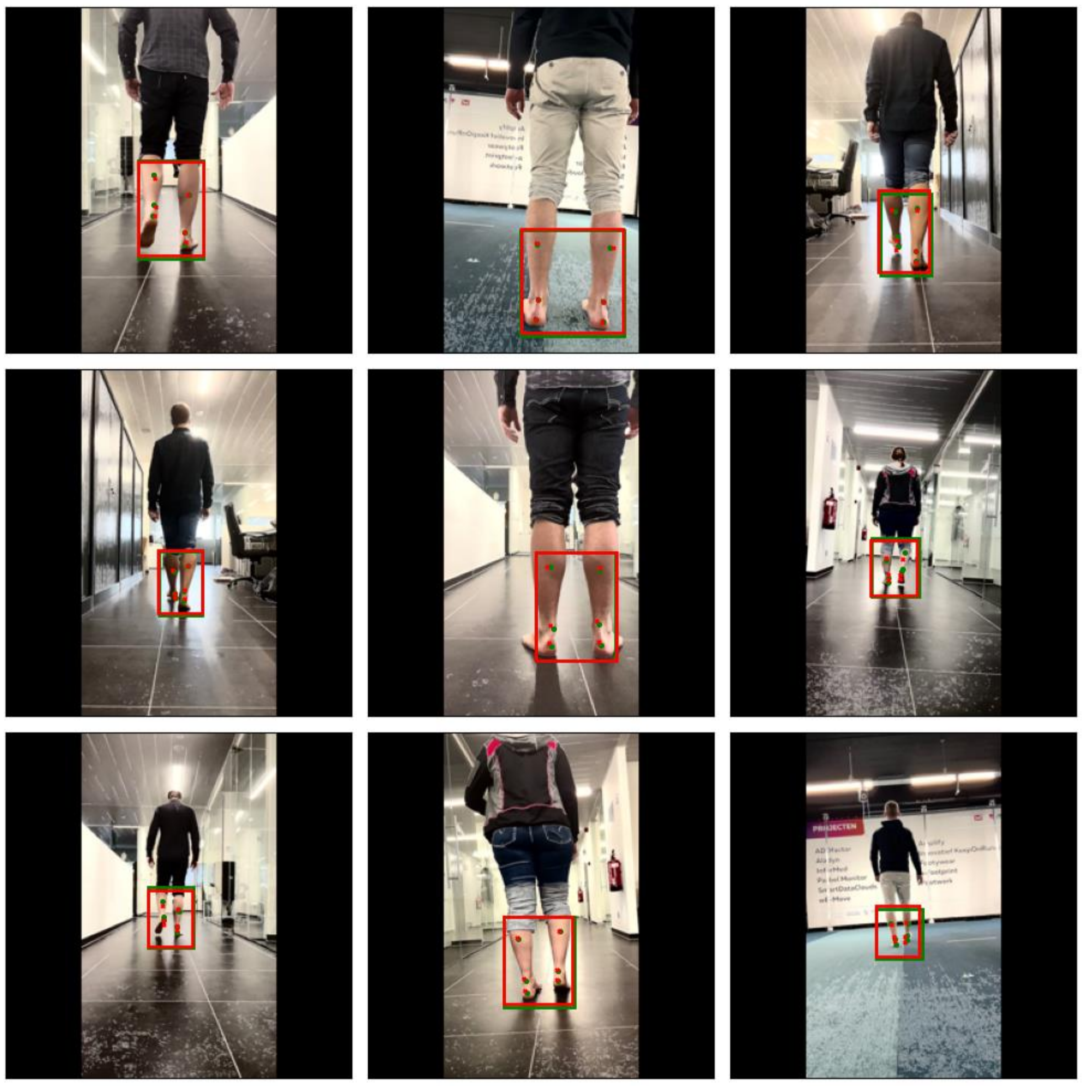
Model multihead_model_test10



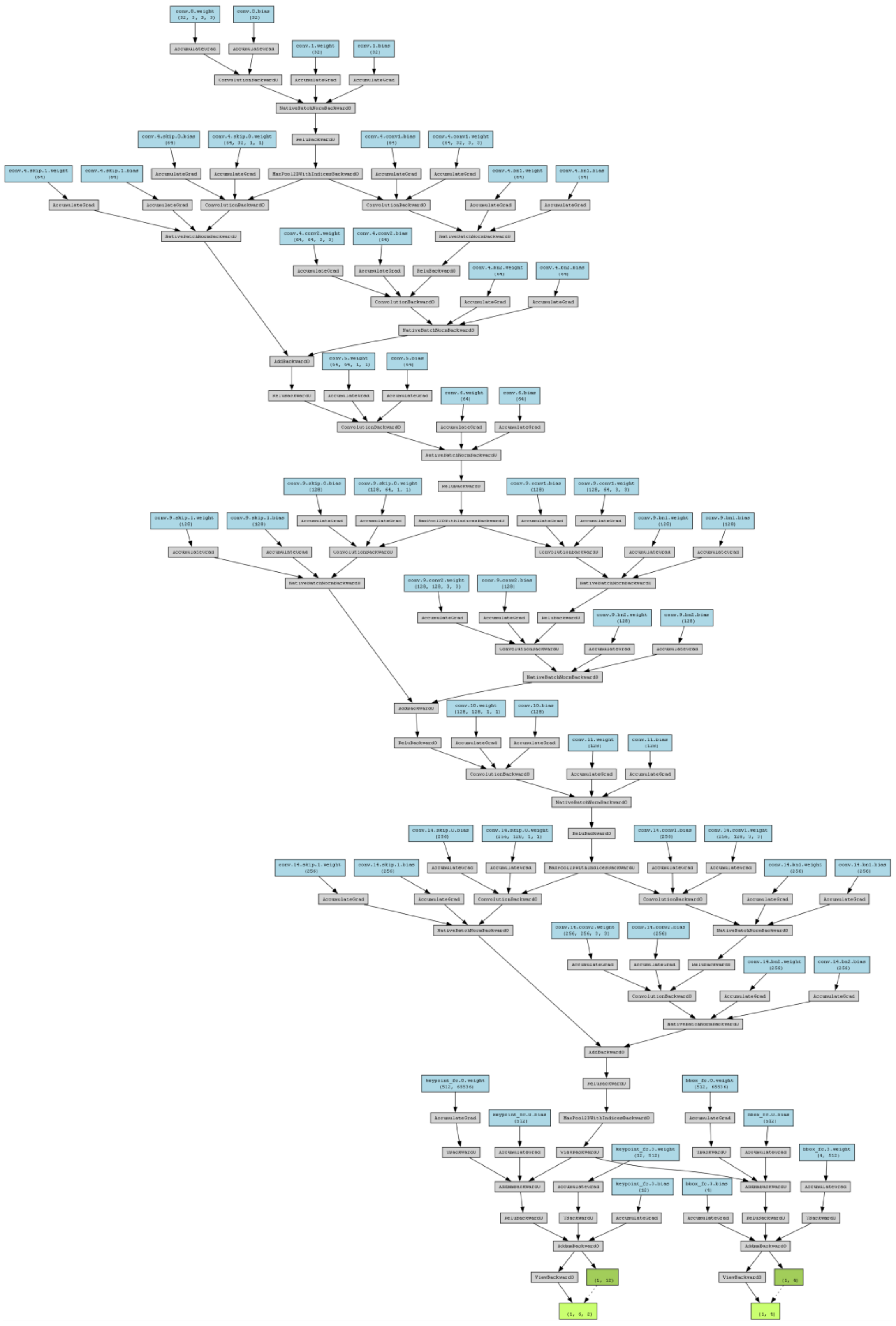




The multiheaded model training presented shows varying behavior across stages, reflecting differences in bias and variance. In stage 1, the model exhibits a steady decrease in both training and validation loss for bounding box head, indicating effective learning and generalization, albeit with occasional oscillations in validation loss, suggesting a moderate variance. By the end of this stage, both losses stabilize, indicating a good balance between bias and variance. In stage 2, the validation loss of the keypoint head displays high variance, oscillating significantly until epoch 18, after which it started to stabilize suggesting good generalization. In stage 3, the training loss of both heads decreases alongside validation loss, suggesting reduced variance. However, there was a slight presence of fluctuation in validation losses in both head at the start of the stage 3. This multi-stage approach demonstrated an interesting way of training a multi-headed model which achieved a good bias-variance trade-off.



Learning from the Detectron2's approach and implementing a custom version of a multi-headed model has paid off. Based on the sanity check and the evaluation matrices, it clearly became the best custom model.



This architecture is designed for localization tasks, employing residual blocks and 1x1 convolutional bottleneck layers in the same way as the keypoint model. The obvious difference it in the final stage which consists of a dual-head output, designed to simultaneously predict bounding boxes and keypoints, making the model well-suited for applications such as this one where the bounding box knowledge helps the predict the keypoints.

4 WORKFLOWS

There are two types of workflows mentioned in this section, development workflow and the operational workflow. The development workflow refers to the sequence of tasks and processes undertaken to conceptualize, design, and build a functional prototype. For AI development this would include collecting and preparing data, selecting and training models, implementing the testing of the model, and validating outcomes. On the other hand, the operational workflow focuses on how the completed system functions during deployment. It describes the runtime behavior of the application, typically including data input, data processing, and output generation such as alerts, reports, or visualizations. While the development workflow is concerned with feasibility and iteration, the operational workflow emphasizes reliability, efficiency, and user interaction. Most of the time during the internship was focused on researching and working on a development workflow. Operational workflow was mostly a part of the additional task towards the end of the internship, since it is connected to the test application.

The following sections will describe in detail the implementation of development workflow for pretrained models, development workflow for custom models, and operational workflow of the test application.

4.1 Development workflow for pretrained model

Developing a Detectron2 model follows a structured workflow that ensures robust training and evaluation. This process involves setting up the framework, preparing data, training the model, and finally evaluating its performance. It does not focus on choosing the which pretrained model to use since the pretrained model was decided on in a previous section. It is also important to remember that developing an AI model is an iterative process during which you slowly figure out what improves the results.

4.1.1 Setup of Detectron2

The setup of Detectron2 begins with installing the necessary dependencies and configuring the environment. The two most important installations for Detectron2, which are Python and PyTorch since it was developed with those tools. It is also important to install all the necessary libraries. Here are a few useful sources to setup the development environment. (*Installation — Detectron2 0.6 Documentation*, n.d.) (Facebookresearch, n.d.-a)

On the topic of environments, it is always beneficial to work with a virtual environment such as venv or a docker container. This helps maintain consistency across different setups making collaborations easier. It is recommended to use Python's virtual environment (venv) and maintain an up-to-date configuration file.

In the case of this internship, there is a file for this section which is called `` `*Detectron2/README.md*` `` and can be found in the handover folder.

Common challenges with setting up a Detectron2 development environment include dependency conflicts, incorrect PyTorch versions, and CUDA compatibility errors. Ensuring the correct versions and testing the installation minimizes these risks.

4.1.2 Data collection & preparation

Preparing a dataset is crucial for training a Detectron2 model. Starting with image collection. Gathering a diverse set of images for the training data is an important step, which will help the model generalize better, making it more robust. Next would be image labeling. Detectron2 uses a COCO JSON format to store information about the datasets it uses, so the annotations need to fit that format. Tools such as COCO Annotator, makesense, or LabelMe generate the required format and are relatively easy to use. Do not forget to organize images and annotation files into the necessary directory structure to be properly loaded into Detectron2. As a part of preprocessing, ensure images have uniform resolution, clean data inconsistencies, and use augmentation on the dataset for robust training data.

In the case of this internship, the scripts utilized for this section can be found in the `*Dataset*` directory in the handover folder.

4.1.3 Model training

Training a model involves multiple critical steps. Detectron2 makes it relatively easy with many prebuild tools. For example, for data loading use the Detectron2's built-in dataset loaders to read images and annotations. A good idea is to always visualize the data that was loaded to make sure it is correct. Again, setting up a training loop for Detectron2 is relatively simple because of it prebuild Trainer tools. An important part of training is the validation and monitoring of the training through loss metrics and validation datasets to identify problems and to fix them in later iterations. It is important not to forget to save and store the trained model and the data developed during training. Also take notes on changes that you have made to the model, so that you are able to track progress.

In the case of this internship, the scripts utilized for this section can be found in the `*Detectron2/Research*` subdirectory in the handover folder.

4.1.4 Model testing & evaluation

Evaluating the trained model determines its possible effectiveness and reliability if it go deployed into an application. There are several ways to test and evaluate a model. A frequently used method is to calculate the discrepancies in keypoint positions and resulting angle deviations. You can also use different performance metrics such as Mean Average Precision (mAP), MSE, RMSE, or AP calculations. There is plenty to choose from and it all depends on the type of model you are trying to evaluate. In the case of Detectron2, it uses AP, AP50, and AP75 for both bounding box and keypoint predictions. Next common way to test models is to display predictions and true labels on test images to visually inspect detection quality. This is also known as a sanity check.

After finishing the model testing and evaluation, figure out how to improve the model. Usually, the tests should have uncovered an issue that you might be facing. So next would by adjusting the model and retraining it until the errors reach acceptable thresholds.

In the case of this internship, the scripts utilized for this section can be found in the `*Detecron2/Visualization*` subdirectory in the handover folder.

4.2 Development workflow for custom model

Developing a custom AI model follows a similar structured approach as Detectron2 but offers more flexibility in defining network architectures, dataset structures, and training strategies. Unlike pretrained models, a custom model requires additional steps in defining and implementing fundamental components such as data handling, training loops, and evaluation metrics.

Same as the development of Detectron2 solution, developing a custom model is an iterative process during which you test different network architectures, training approaches, or hyperparameter configurations. Continuous iteration through model refinements ensures steady improvement.

In the case of this internship, all subsections of 4.2 are part of scripts that can be found in the \`*Custom/Research*\` subdirectory in the handover folder.

4.2.1 Setup of PyTorch

Setting up PyTorch is relatively straightforward. Begin by installing PyTorch using the recommended installation guide for your system. (*Get Started*, n.d.)

Since PyTorch supports multiple environments, it is advisable to use Python's virtual environment (venv) to ensure dependency management and prevent conflicts. Additionally, maintaining a requirements file (requirements.txt) helps streamline installation across different setups.

For CUDA compatibility, confirm that the correct CUDA version is installed to enable GPU acceleration, which significantly boosts training speed. Simply running \`*torch.cuda.is_available()*\` in your Python code verifies if PyTorch detects the GPU.

4.2.2 Data collection & preparation

Data collection remains the same as in Detectron2, gathering diverse and well-structured training data to improve model generalization. However, for a custom model, dataset definitions must be explicitly created.

In order to save time, the goal was to use the same datasets as for Detectron2, which utilizes COCO dataset format. A custom dataset class was implemented to efficiently load the COCO style dataset used for training Detectron2.

4.2.3 Model development

Developing the model starts with defining the Model class, which encapsulates the architecture of the neural network. This involves structuring the layers of the network or implementing the forward propagation function to handle input transformations.

It is a good idea to keep modularity in mind. By using inheritance to create a simple versioning system for model updates, making iterative improvements easier.

4.2.4 Model training

Training the custom model involves designing an effective training function. A critical aspect of this is monitoring loss values and validation performance to ensure steady improvements in the training process.

In the case of multi-headed models, implementing a multi-stage training can be particularly useful to separately refine the individual output heads of the model.

Training steps usually include:

1) Training Phase
    - Reset gradients and run the model on training images
    - Calculate loss and update weights
    - Track gradient values and learning rate
    - Repeat for all training batches
    - Compute average training loss
2) Validation Phase

    - Switch to evaluation mode (no gradient updates)
    - Run the model on validation images and compute loss
    - Repeat for all validation batches
    - Compute average validation loss
3) Adjust Learning Rate and Log Results
    - Update the learning rate based on validation loss
    - Log epoch details (training/validation loss, learning rate, and gradient values)

4.2.5 Model testing & evaluation

Similar to Detectron2, model evaluation is essential to assess its performance. The evaluation function follows the same structured approach:

1) Running predictions on test datasets and comparing them with ground truth labels.

2) Measuring performance using relevant metrics such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Mean Average Precision (mAP).

3) Conducting visual inspections by overlaying predicted results on test images as a sanity check.

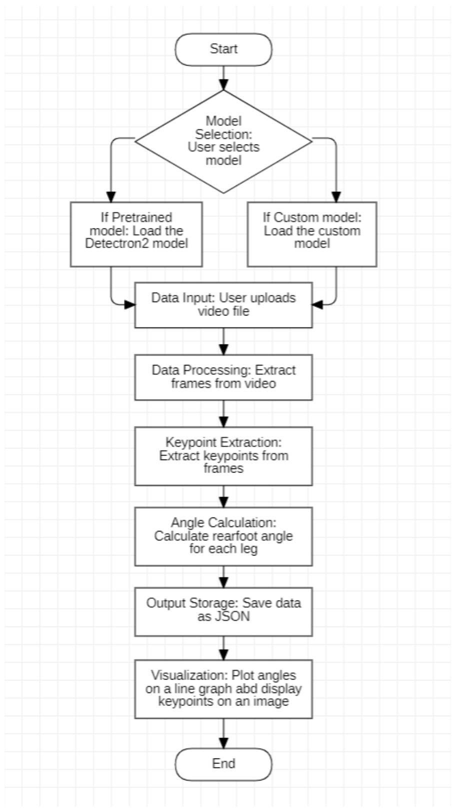
4.3 Operational workflow

The primary objective of the operational workflow is to outline the sequence of operations involved in processing video data, predicting keypoints, calculating angles, and visualizing the results. By structuring the process into clear and manageable steps, the workflow enhances the efficiency and reproducibility of the analysis. Furthermore, it ensures that users can seamlessly interact with the application through a user-friendly graphical interface made in Streamlit.

In the case of this internship, all subsections of 4.3 are part of scripts that can be found in the `Test_Application` directory in the handover folder.

On the right side, you can see the overall operational workflow of the application as a flowchart. The following subsection will go over the details of the flowchart.

Here is a short demo video of the test application. For better visualization of the workflow. (MatejB, 2025)


Start
Model Selection: User selects model
If Pretrained model: Load the Detectron2 model
If Custom model: Load the custom model
Data Input: User uploads video file
Data Processing: Extract frames from video
Keypoint Extraction: Extract keypoints from frames
Angle Calculation: Calculate rearfoot angle for each leg
Output Storage: Save data as JSON
Visualization: Plot angles on a line graph abd display keypoints on an image
End

4.3.1 Model selection

In the first step in the workflow, the user selects the desired model from a drop-down menu in the sidebar. The application provides two model options: a custom PyTorch model and a Detectron2 model. Upon selection, the chosen model is loaded into the system, and the configuration is set according to predefined parameters to ensure consistent predictions.

4.3.2 Data input

The next step in the workflow involves data input, where the user uploads a video file through the Streamlit interface. The application supports a range of common video formats, including MP4, AVI, MOV, and MKV, to accommodate diverse user requirements. Once uploaded, the video file is stored in a designated directory, prepared for frame extraction.

4.3.3 Data processing

Following the upload, the video is processed to extract individual frames. This step is performed using OpenCV, a computer vision library that efficiently handles video manipulation. Each frame is saved as a separate image file in a structured directory. Subsequently, these frames are prepared for keypoint extraction by ensuring uniformity in size and format, thus optimizing them for model inference.

4.3.4 Keypoint extraction

The next step involves keypoint extraction from each video frame. The selected model analyzes the images, predicting the positions of keypoints that correspond to markers placed on the patient by the doctor. These keypoints are then resized to match the original image dimensions and are classified into left and right leg clusters using the K-means clustering algorithm. This classification is essential for calculating the angles between segments accurately.

4.3.5 Angle Calculation

Once the keypoints are identified and classified into correct legs, the application calculates the angles formed by the segmented points. The angle calculation method takes into account the spatial configuration of the points and computes the angles. Special handling ensures that angles greater than 180 degrees are correctly adjusted to provide accurate biomechanical measurements.

4.3.6 Output Storage

After the angles are calculated, the results are stored. The calculated angles and keypoint coordinates are saved as JSON files within a structured directory. Users can select a specific output file from the Streamlit interface to view and analyze the corresponding data.

4.3.7 Visualization

The final step in the workflow is the results being visualized through interactive graphs and annotated images. The application uses Plotly to generate dynamic graph that plots the angles over time, enabling users to track changes throughout the video. Additionally, the images are displayed with highlighted keypoints and labelled angles, providing a clear visual representation of the analysis outcomes.

5 CONCLUSION

The aim of this internship project was to research and develop a proof-of-concept for automated rearfoot angle tracking using video images. The intended use case was for clinical environments, where the system could assist clinicians in making rearfoot angle measurements that could be used for designing personalized shoe inserts. The project spanned thirteen weeks and involved design, implementation, and comparison of two fundamentally different approaches. One using a pretrained model (Detectron2) and another using custom-built deep learning models implemented in PyTorch.

The project began with a thorough analysis of available pretrained models for pose estimation. After evaluating licensing, performance, ease of use, and community support. Detectron2 was chosen due to its suitable licensing, ease of use and relatively good community support. A range of training configurations of the model and using different bounding box sizes and augmentation strategies on the dataset were tested, leading to the selection of output_5 (name of the model) as the best-performing Detectron2 model.

After investigating pretrained models, significant effort was invested in developing custom models from scratch using PyTorch. These models included keypoint detection models, localization modules, and a final multi-headed model that merged both functions into a single network. Architectural experiments explored residual blocks, bottleneck layers, dropout regularization, different training loss functions (MSE, RMSE, Smooth L1), and preprocessing strategies such as histogram equalization, CLAHE, edge detection overlays, and noise reduction. Out of all the experiments, the most accurate custom model was multihead_model_test10. It employed a shared feature extraction backbone with two separate heads (bounding box prediction and keypoint detection), and training was performed using a multi-stage strategy (first training each head separately, then together). This model showed excellent performance in both training and validation, with minimal overfitting and good generalization ability on unseen test data.

To demonstrate the model in a practical setting, a test application was developed using Streamlit. This interface enables clinicians to upload videos, choose between different models, process the footage images, and visually inspect angle predictions. The application includes interactive graphs (built using Plotly) and visualizations with overlaid keypoints.

5.1 Possible improvements

Improving the performance and robustness of the keypoint detection model can be achieved through multiple avenues. Below are some suggestions that should be considered for potential improvements and future research directions.

5.1.1 Custom model improvements

Exploring Long Short-Term Memory (LSTM) and Transformers Architectures could improve spatial-temporal feature learning, especially for tasks involving objects in motion with relatively simple movement patterns.

Implementing Float16 mixed precision training can optimize training efficiency, reduce memory usage, and speed up computations. The integration of mixed precision training, as demonstrated in FastAI 's implementation, could improve computational performance while maintaining accuracy. (Fastai, n.d.)

5.1.2 Data collection improvements

By collecting more varied data such different lighting conditions, perspectives, and subjects with different racial backgrounds would enhance model generalization and robustness. Collecting additional samples with diverse backgrounds and ethnic races can help mitigate biases and improve real-world applicability.

5.1.3 Data augmentation strategies

While color shift augmentation was explored, its impact on model inclusivity remains uncertain. Future tests could quantify its effectiveness more rigorously and refine the implementation to improve performance across different races and varied lighting environments.

Implementing image tilt augmentation may increase robustness against perspective distortions, benefiting keypoint detection in cases where input images are not perfectly aligned with the patient.

5.1.4 Data preprocessing strategy

Background subtraction is an interesting preprocessing step that could help isolate the relevant subject from unnecessary environmental noise. There are two common approaches. A traditional method using techniques such as Gaussian Mixture Models or OpenCV's background subtraction methods can effectively remove static backgrounds. Alternatively, a deep learning method using U-Net-based segmentation or semantic segmentation models can help distinguish foreground and background with higher precision, especially in complex environments, but increases the computation requirement drastically.

5.2 Evaluation

This section evaluates the outcomes of the internship project in relation to the objectives defined in the project plan. The assignment's core aim was to design and implement a proof-of-concept workflow for automated rearfoot angle tracking using video images. Additionally, if there was time spare, the plan proposed the development of a test application based on the most successful model. The evaluation is structured around these key objectives, reflecting on whether they were achieved, to what extent, and with what results.

The primary objective was to create a complete proof-of-concept workflow capable of estimating the rearfoot angle from video footage by detecting specific keypoints on the legs. This goal was fully accomplished. Two approaches were developed: one based on the pretrained Detectron2 model and one based on custom models developed in PyTorch. Both of the final development workflow successfully results in creating a functional keypoint detecting AI model. It is also important to mention that both methods resulted in a commercially viable. Detectron2 runs under an Apache-2.0 license and developing a custom model lets you use it however you want.

In the project plan, development of a test application was an optional extension, to be done after the main objective was completed within schedule. This optional goal was also accomplished. A comprehensive test application was developed using Streamlit, offering a complete end-to-end solution. Users can select models, upload video footage, initiate processing, and review results via interactive graphs and frame-by-frame visualization of the video. The application provides an intuitive tool suitable for clinical demonstration before actual development of user ready application.

6 DICTIONARY FOR SHORTCUTS

Conv3x3: Convolution with a three-by-three kernal

Conv1x1: Convolution with a one-by-one kernal

BN: Batch Normalization – Normalizes activations across a mini-batch to stabilize training

ReLU: Rectified Linear Unit – Activation function that sets negative values to zero

MaxPull: Max Pooling – Downsampling operation that selects the maximum value in a region

SkipConnection: Connection that bypasses layers to help gradient flow and mitigate vanishing gradients

RB: Residual Block – A building block in deep networks that uses skip connections

FE: Feature Extraction – Extracting features from a pretrained neural network architecture like ResNet

MSE: Mean Squared Error – Loss function measuring average squared differences between predictions and targets

RMSE: Root Mean Squared Error – Square root of MSE, providing error in the same scale as original data

L1 Loss: Least Absolute Deviations – Measures absolute differences, making it robust to outliers

L2 Loss: Least Squares Error – Measures squared differences, penalizing larger errors more heavily

Smooth L1 Loss: Loss function that transitions from L1 to L2 loss for robustness to outliers

IoU: Intersection over Union – Metric measuring overlap between predicted and ground-truth bounding boxes

AP: Average Precision – If it is followed by a number, it represents the threshold, meaning for AP50, the prediction must align at least 50% with the actual truth to count as correct.

mAP: Mean Average Precision

7 Reference List

Ultralytics. (2025, February 26). *YOLO11 🚀 NEW*. https://docs.ultralytics.com/models/yolo11/

Ultralytics. (n.d.). *GitHub - ultralytics/ultralytics: Ultralytics YOLO11 🚀*. GitHub. https://github.com/ultralytics/ultralytics

Facebookresearch. (n.d.). *GitHub - facebookresearch/detectron2: Detectron2 is a platform for object detection, segmentation and other visual recognition tasks.* GitHub. https://github.com/facebookresearch/detectron2

Open-Mmlab. (n.d.). *GitHub - open-mmlab/mmpose: OpenMMLab Pose Estimation Toolbox and Benchmark.* GitHub. https://github.com/open-mmlab/mmpose

DeepLabCut — the Mathis Lab of Adaptive Intelligence. (n.d.). The Mathis Lab of Adaptive Intelligence. https://www.mackenziemathislab.org/deeplabcut

DeepLabCut. (n.d.). *GitHub - DeepLabCut/DeepLabCut: Official implementation of DeepLabCut: Markerless pose estimation of user-defined features with deep learning for all animals incl. humans*. GitHub. https://github.com/DeepLabCut/DeepLabCut

Cmu-Perceptual-Computing-Lab. (n.d.). *GitHub - CMU-Perceptual-Computing-Lab/openpose: OpenPose: Real-time multi-person keypoint detection library for body, face, hands, and foot estimation*. GitHub. https://github.com/CMU-Perceptual-Computing-Lab/openpose?tab=readme-ov-file

Mvig-Sjtu. (n.d.). *GitHub - MVIG-SJTU/AlphaPose: Real-Time and Accurate Full-Body Multi-Person Pose Estimation&Tracking System*. GitHub. https://github.com/MVIG-SJTU/AlphaPose

Lin, T., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., & Dollár, P. (2014, May 1). *Microsoft COCO: Common Objects in context*. arXiv.org. https://arxiv.org/abs/1405.0312

PyTorch. (n.d.). PyTorch. https://pytorch.org/

TensorFlow. (n.d.). *TensorFlow*. https://www.tensorflow.org/

Team, K. (n.d.). *Keras: Deep Learning for humans*. https://keras.io/

Welcome to fastai – fastai. (n.d.). Fastai. https://docs.fast.ai/

JAX: High performance array computing — JAX documentation. (n.d.). https://docs.jax.dev/en/latest/

Apache MXNet. (n.d.). Apache MXNet. https://mxnet.apache.org/versions/1.9.1/

Wikipedia contributors. (2025, May 21). *Canny edge detector*. Wikipedia. https://en.wikipedia.org/wiki/Canny_edge_detector

NumPy – Understanding tan(), arctan(), and arctan2() functions (6 examples) - Sling Academy. (n.d.). https://www.slingacademy.com/article/numpy-understanding-tan-arctan-and-arctan2-functions-6-examples/

Lad, R. (2024, November 18). An introduction to Detectron2 - Ravina Lad - Medium. *Medium*. https://medium.com/@ravina.lad01/an-introduction-to-detectron2-1b15e2f39b19

Installation — detectron2 0.6 documentation. (n.d.). https://detectron2.readthedocs.io/en/latest/tutorials/install.html

Facebookresearch. (n.d.-a). *Can I install detectron2 on Win10? · Issue #9 · facebookresearch/detectron2*. GitHub. https://github.com/facebookresearch/detectron2/issues/9#issuecomment-1123096793

Get started. (n.d.). PyTorch. https://pytorch.org/get-started/locally/

MatejB. (2025, May 21). *internship_demo_video* [Video]. YouTube. https://www.youtube.com/watch?v=p07WWnMXMXk

Fastai. (n.d.). *fastai/fastai/callback/fp16.py at main · fastai/fastai*. GitHub. https://github.com/fastai/fastai/blob/main/fastai/callback/fp16.py