TeCNNis: A Dual-CNN Pipeline for Real-Time Tennis Shot Landing Prediction

Candidate Numbers: 50537, 44158, 39706, 50615

Abstract

This project proposes a deep learning pipeline using two custom Convolutional Neural Networks (CNN) for predicting the outcome of a tennis shot, before the shot has concluded. Our models are trained on a publicly available tennis shot dataset featuring labeled shot trajectories from indoor and outdoor tennis videos. The first stage of the system uses a CNN to identify the likelihood that the ball is hitting the racket in a specific video frame. The second stage uses another CNN to process a short sequence of frames around the identified hit frame to predict the ball's landing coordinates on the court. We explore different unique methods to train such a system, and evaluate performance against pre-trained CNNs. We find that this system has significant potential to be used in live tennis betting markets, or in other applications such as automated line-calling.

1 Introduction

The intersection of sports analytics and recent developments within Artificial Intelligence (AI) present opportunities for generating informational value, especially in near real-time scenarios. Sports betting markets work on outcomes of events rather than predictions, providing a significant opportunity for application in predictive AI. Exploiting this inefficiency, however, requires a highly-tailored, highly-optimised low-latency solution.

Recent years have seen a transformative increase in the prevalence and capabilities of AI especially in tasks such as image processing - notably the development and wide-spread usage of Convolutional Neural Networks (CNNs). These computer vision models have excelled in image-recognition, classification and object detection thanks to their inherent ability to learn and dynamically extract relevant key features within images, but we believe the application of these models for predictive analysis within real-time sports matches to forecast outcomes is a relatively nascent and under-explored area.

In this paper, we introduce a bespoke deep learning framework for this challenging task. Our framework utilises a dual-CNN system: an initial CNN dedicated to accurately identifying the moment of ball-racket impact from the video stream, and a second CNN designed to process a sequence of frames immediately surrounding the impact frame and output a coordinate prediction of the landing point of that shot.

The main objective of our project is to create and evaluate an AI system for the purpose of accurately predicting whether a tennis shot will land in or out of bounds prior to it actually landing. This pre-landing predictive power is designed to yield an informational advantage in order to hopefully identify, and take advantage of, lagged wagering opportunities across betting markets.

This paper outlines the fundamentals of the prediction system, the training process, various experiments, and the finally the results achieved. We also explore ways in which this research could be developed, to create a fully operational predictive tennis pipeline for use in commercial settings.

2 Related Work

Computer vision applications in sports analytics, particularly tennis, represent a growing area of research, although real-time predictive analysis of shot outcomes remains relatively less explored.

Prior work like TrackNet (Yastrebov [2019]) has successfully employed CNNs for tracking the tennis ball and estimating its trajectory across the court. While relevant in demonstrating the use of CNNs for ball movement analysis, TrackNet's primary focus was post-hoc trajectory reconstruction, often optimised for challenging, low-resolution video streams. Our approach diverges significantly by aiming for *predictive* analysis - specifically forecasting the precise landing point coordinates *before* the bounce occurs - using a distinct dual-CNN pipeline potentially suited for clearer, broadcast-style top-down views.

Research on classifying tennis hit types (e.g., forehand, backhand, serve) has also been conducted. For instance, a Stanford class project (Chatterjee et al. [2020]) explored models including LRCNN (incorporating LSTMs for temporal analysis) and EvaNet for classifying different shots based on player motion and technique. This differs fundamentally from our objective, which is a *regression* task focused on predicting continuous (X, Y) landing coordinates rather than discrete shot categories. Our method implicitly captures short-term temporal context through stacked frame inputs to the second CNN, offering an alternative to explicit recurrent architectures like LSTMs used for classification tasks.

Our specific implementation and evaluation are built upon the publicly available top-view tennis shot dataset curated by Lai et al. [2024]. This dataset, designed for player performance analysis, provides the essential labeled landing coordinates paired with video sequences, which is crucial for training and validating our landing point regression model.

In summary, while drawing on concepts from ball tracking and leveraging deep learning techniques seen in shot classification, our project targets the specific, less-explored niche of real-time, pre-bounce landing point prediction using a novel dual-CNN pipeline.

3 Dataset and Preprocessing

3.1 Dataset Description

The framework is developed and evaluated using the "Tennis Shot Side-View and Top-View Data Set for Player Analysis in Tennis" dataset. This dataset comprises 472 video clips, evenly split between side-view and top-view perspectives (236 clips each). It is categorised into two main environments: Outdoor (clay court) and Indoor (leisure centre type courts), with each being further divided by shot type: straight shot and cross-court shot. The dataset includes videos capturing the player's swing and provides the landing point coordinates for each shot, recorded in CSV files. These landing coordinates are the distance from the closest doubles sideline and the baseline, with file names indicating the field category and shot type. This comprehensive dataset provides a robust basis for training and testing our models.

The source data set was published in May 2024 as a "valuable asset to the tennis community". It is ideal for this project, as it contains numerous tennis shot videos paired with their respective landing coordinates. For this project we used the top-down views only as this is the standard recording angle for tennis matches, meaning the models will likely generalise better on out-of-sample videos. We preprocessed the data by splitting the videos into frames, and uploading this to Kaggle for easy downloading onto Google Colab. We also manually annotated the exact hit frame for each video to allow supervised training of CNN1, stored in `hit_frames.csv`.

3.2 CNN1 Data Preparation (Hit Frame Identification)

Data for CNN1 involved assigning weights to frames loaded from `hit_frames.csv`. A custom function identified true hit frames (`is_hit_frame == 1`) and assigned them weight 1.0. Neighboring frames within a sequence window received linearly decaying weights (decay factor 0.3) to handle potential ball obscurity by the racket and provide temporal context, improving generalisation. Frames outside this window received weight 0.0. This notably transforms the task into regression, which we considered advantageous for generating a continuous signal rather than binary classification. To address data imbalance from many zero-weight (non-hit) frames, the dataset was balanced by undersampling these frames at a 4:1 ratio (determined by grid search) relative to positively weighted frames. This balanced dataset was then split into 80%/10%/10% train/validation/test sets using `sklearn.model_selection.train_test_split`, stratified by positive/zero weight where possible. Image dimensions were standardised to 224x224.

3.3 CNN2 Data Preparation (Landing Point Prediction)

Landing coordinate data from four condition-specific CSVs was combined, linked to video sequences via a unique `ShotID` generated from filenames and indices. Raw coordinates (distances to sideline/baseline in meters) were normalised to $[0,1] \times [0,1]$ using the `map_coordinates` function (Algorithm 1). This function accounts for shot type ('Straight'/'Cross') relative to the left sideline (x=0) and uses standard court dimensions (`DOUBLES_COURT_WIDTH_M = 10.97`m, `HALF_COURT_LENGTH_M = 11.89`m), with the Y coordinate increasing towards the net.

Algorithm 1 Coordinate Mapping Function (`map_coordinates`)

```
Require: Distance to closest doubles sideline $d_{side}$ (m), Distance to receiver's baseline $d_{base}$ (m),
    Shot type $type \in \{$'Straight', 'Cross'$\}$, Court width $W_{court}$ (m), Half court length $L_{court}$ (m)
Ensure: Normalised coordinates $(x_{norm}, y_{norm}) \in [0,1] \times [0,1]$, or None if inputs invalid
 1: if $type \notin \{$'Straight', 'Cross'$\}$ or $W_{court} \leq 0$ or $L_{court} \leq 0$ then
 2:     return None
 3: end if
 4: $y_{abs} \leftarrow d_{base}$
 5: $y_{norm} \leftarrow y_{abs}/L_{court}$
 6: if $type ==$ 'Straight' then
 7:     $x_{abs} \leftarrow W_{court} - d_{side}$
 8: else if $type ==$ 'Cross' then
 9:     $x_{abs} \leftarrow d_{side}$
10: end if
11: $x_{norm} \leftarrow x_{abs}/W_{court}$
12: $x_{norm} \leftarrow \max(0.0, \min(x_{norm}, 1.0))$
13: $y_{norm} \leftarrow \max(0.0, \min(y_{norm}, 1.0))$
14: return $(x_{norm}, y_{norm})$
```

Input sequences for CNN2 were generated by first identifying the true hit frame (weight=1.0). For each video, a sequence of `N_FRAMES_SEQUENCE = 9` frames, centered on the hit frame's index, was extracted from the chronologically sorted frame list. These sequences were paired with their corresponding normalised landing coordinates (NormX, NormY) retrieved via `ShotID`. This formed the CNN2 dataset, subsequently split into 80%/10%/10% train/validation/test sets, stratified by environment where possible.

3.4 Data Augmentation

In order to improve generalisation (reduce overfitting) and to create a larger data set for training we randomly augmented our training data set in the following ways: Horizontal flip, Court colour change, and Keystone/ perspective shift. Each augmentation occurs probabilistically with the following probabilities: **Horizontal flip**: 0.5; **Colour change**: *None*: 0.33, *Blue*: 0.33, *Green*: 0.33; **Keystone shift**: 6 degrees of keystone shift, equally likely, in any direction. These combined augmentations increased our effective dataset size by an order of approximately 70 - a substantial increase. This has the benefit of mitigating overfitting and increasing generalisation on unseen data but dramatically increased the number of epochs required to achieve the same training performance.

An example batch of frames with augmentations applied can be found in Figure 1.

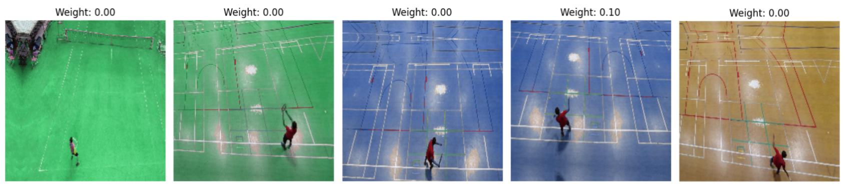


Figure 1: Sample Batch of Frames

4 Methodology

4.1 Overall Framework

Our approach utilises two Convolutional Neural Networks (CNNs) referred to in the following as CNN1 and CNN2. CNN1 takes a frame from a tennis shot video and returns a confidence score based on how likely it is the ball is being hit in that frame. For each video, the frame with the highest likelihood of being the hit frame, as determined by CNN1 is fed into CNN2, which regresses the frame onto a coordinate point, predicting the landing spot of the tennis shot. This architecture facilitates live shot prediction - CNN1 can view frames sequentially until it reaches a threshold likelihood, and then this frame can be passed live into CNN2, thereby potentially predicting the landing location before it occurs.

The project is written in Python using PyTorch for neural net architecture and OpenCV for video processing.

4.2 Model Architectures

4.2.1 CNN1: Hit Frame Regressor (`HitFrameRegressorFinal`)

CNN1 (Figure 2) takes a single RGB image of size $(3, 224, 224)$ as input. The architecture, determined through hyperparameter search (see Section 5), consists of four sequential convolutional blocks followed by a fully connected head. Each convolutional block applies two sets of 3×3 convolutions (with stride 1, padding 1), each followed by Batch Normalisation (`nn.BatchNorm2d`) and a ReLU activation function (`nn.ReLU`). A 2×2 Max Pooling layer (`nn.MaxPool2d`) with stride 2 concludes each block, progressively reducing spatial dimensions. The number of filters in the convolutional layers for the four blocks are specified by `block_filters = (32, 32, 64, 64)`. The output feature map from the final convolutional block is flattened and passed through a fully connected layer with `fc_size = 256` hidden units and ReLU activation, followed by a Dropout layer with a rate of `dropout_rate = 0.5`. The final layer is a single linear neuron producing the regression score (hit likelihood).

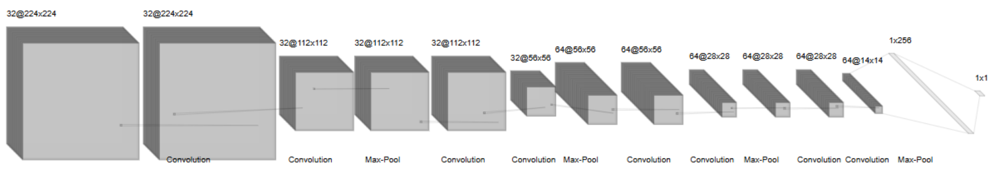


Figure 2: Architecture diagram of CNN1 (`HitFrameRegressorFinal`)

4.2.2 CNN2: Landing Point Predictor (`LandingPointCNN`)

CNN2 (Figure 3) receives a sequence of $N = 9$ frames stacked along the channel dimension, resulting in an input tensor of shape $(N \times 3, 224, 224)$. The architecture employs a series of convolutional and pooling layers to extract spatial features. It includes four main convolutional stages. For example, the first stage uses a 7×7 convolution (`nn.Conv2d`) with stride 2 and padding 3, followed by BatchNorm and ReLU, and then a 3×3 Max Pooling layer with stride 2. Subsequent stages use smaller kernels (e.g., 5×5, 3×3) with stride 1, interspersed with BatchNorm, ReLU, and Max Pooling (stride 2). Filter counts increase through the layers: 128, 256, 512, and 512. The output of the final pooling layer is flattened and passed through fully connected layers of sizes 512, and 256 with ReLU activations and Dropout (rate 0.5). The final output layer consists of two neurons corresponding to the predicted normalised x and y coordinates. Crucially, a Sigmoid activation function (`nn.Sigmoid`) is applied to this final layer to ensure the outputs fall within the range $[0, 1]$, matching the normalised target coordinates.

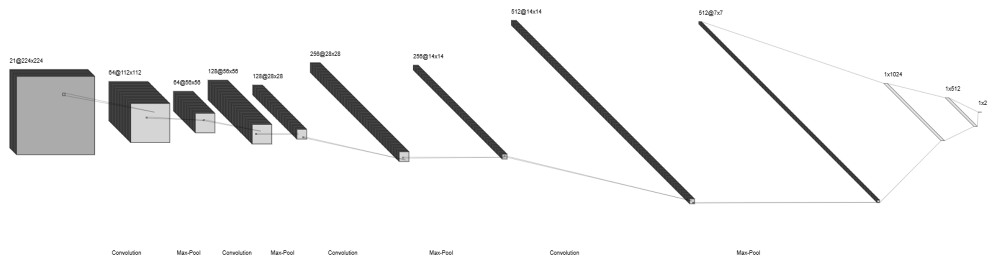


Figure 3: Architecture diagram of CNN2 (`LandingPointCNN`)

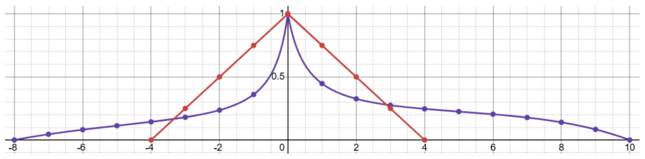


Figure 4: Simple v.s Complex Frame Weighting Functions

4.3 Training Procedures

4.3.1 Standard Training (Separate Models)

Models were trained individually using MSE loss and Adam optimisation with hyperparameters determined via grid search (Section 5). CNN1 was trained to predict target weights assigned by the linear weighting function (LR=1×10^{-4}, Batch=16). CNN2 was trained to predict normalised landing coordinates (x_{norm}, y_{norm}) from 9-frame input sequences centered via the linear weighting scheme (LR=5×10^{-5}, Batch=16). Both training processes were conducted for a maximum of 150 epochs on a GPU, employed early stopping (patience=15, min improvement $\epsilon = 1 \times 10^{-5}$ on validation MSE), and applied data augmentation (Section 3) during the final run. Training results are detailed in Section 6.

4.3.2 Joint Training Framework

To potentially improve performance by allowing models to learn collaboratively and optimise the hit frame identification for the landing prediction task, we explored a joint end-to-end training framework.

In the joint training paradigm, CNN1 and CNN2 were trained end-to-end to learn the hit frame identification and landing prediction tasks collaboratively. The training process utilised the `JointPredictionDataset`, which provides batches containing: (1) a long sequence of frames ($L = 21$, defined by `JOINT_DATASET_CONTEXT_FRAMES`) centered around the true hit frame (reduces computational load by ignoring irrelevant frames); (2) target weights for each frame in the long sequence, calculated using the Bayesian-optimised $h(x)$ function; (3) target normalised landing coordinates; and (4) the index of the true hit frame within the long sequence.

Parameterised Weighting Function: A key difference in the joint training paradigm is in the method for assigning weights to frames. In the initial CNN1, we assigned weights based on a simple weighting function shown in Figure 4 (red line), where weights decay linearly in both directions outwards from the true hit frame.

In the joint training paradigm, we experimented with a fully parameterised weighting function, able to have asymmetrical frames on either side of the hit frame, and tailoring the weight decay to encourage early hit frame identification. The parameterised weighting function (purple line in Figure 4) uses a custom formula: $h(x) = K_{val} \left(\frac{1}{|x| \left\{ x>0 : \frac{N}{R_1}, \frac{N}{R_2} \right\} + D} - S_{val} \right) \exp \left(|x| \left\{ x > 0 : M_1, M_2 \right\} \right)$ where x is the frame distance from the true hit frame, $S_{val} = \frac{1}{N+D}$ and $K_{val} = \frac{D(N+D)}{N}$, R_1 and R_2 control number of frames before and after the hit frame, M_1 and M_2 control tailedness, and N and D control decay magnitude.

This parameterised function yields much greater control in shaping the way in which the model sees the shot temporally, but introduces complexity. To find suitable parameters for this function in the joint model, we utilised Bayesian optimisation on as this is a powerful optimisation tool for when the feasible reason is continuous and the objective function is computationally expensive (further details in Section 5).

Joint Training Process Within each training step, CNN1 processed each frame of X_{long} (reshaped to X'_{long}) to produce hit scores S_{pred}. The CNN1 loss ($\mathcal{L}_{\text{CNN1}}$) was computed as the MSE between S_{pred} and reshaped target weights. The predicted hit index idx_{pred} was determined from the maximum score in S_{pred}. Using this index, a dynamic input sequence X was constructed by selecting $R_2 = 8$ frames before and $R_1 = 10$ frames after idx_{pred} from X_{long}. CNN2 then processed X to predict landing coordinates Y_{pred}. The CNN2 loss ($\mathcal{L}_{\text{CNN2}}$) was computed as MSE between Y_{pred} and Y_{coords}, and a penalty loss ($\mathcal{L}_{Penalty}$) was computed as the L1 distance (MAE) between idx_{pred} and idx_{true}. The total loss was calculated as $\mathcal{L}_{Total} = \mathcal{L}_{\text{CNN1}} + \mathcal{L}_{\text{CNN2}} + 0.1 \times \mathcal{L}_{Penalty}$. Finally, a single Adam optimiser (LR 1×10^{-5}) updated all weights based on the gradient of $\mathcal{L}_{Total}$.

Training ran for a maximum of 150 epochs, again with early stopping, and with a batch size of 16. Data augmentation (Section 3) was applied. Joint training results can be found in Section 6.

5 Experiments

5.1 Experimental Setup

For all of our experiments, a random seed has been used for reproducibility. The experiments were carried out on Google Colab using a NVIDIA T4 GPU.

The training of the CNNs was performed using MSE loss and an Adam optimiser and all other parameters fixed unless otherwise specified. All tuning runs were 5 epochs long and with data augmentation turned off, to allow greater hyperparameter exploration under computational restraints.

5.2 Hyperparameter and Architecture Tuning

Key hyperparameters and model architectures were determined via grid search and Bayesian Optimisation based on validation performance during short training runs. Optimal learning rates were 1e-4 (CNN1) and 5e-5 (CNN2) with a batch size of 16 (limited by computational resources). We selected a 9-frame input sequence for CNN2 (4 frames flanking the hit frame), using a linear weight decay of 0.3 for frames adjacent to the hit in CNN1's training, and a 1:4 hit-to-non-hit frame balance.

Architecture search led to selecting 4 convolutional blocks for both models. CNN1 uses [32, 32, 64, 64] filters per block and a 256-unit fully connected (FC) layer. CNN2 uses [128, 256, 512, 512] filters per block with FC layers of 512 and 256 units. A dropout rate of 0.5 was applied. For the joint training's complex weighting function, parameters were found via Bayesian Optimisation targeting CNN1 loss, a necessary simplification discussed further in Section 7.

5.3 Pretrained models

To allow us to compare the performance of our custom architecture, we fine-tuned a selection of general image classification models available on Hugging Face(Hugging Face [2024]).

For hit frame regression, we selected ViT-Base (Dosovitskiy et al. [2021]) for its transformer architecture, hypothesising its global attention could better capture frame-wide impact context; ResNet-50 (He et al. [2016]) was chosen as a deep CNN baseline strong at hierarchical feature localisation for the contact point; and we chose ConvNeXt-Tiny (Liu et al. [2022]) as a modern, efficient CNN to test performance on fine-grained hit details.

For landing point prediction we tested ResNet-18 and ResNet-34 (He et al. [2016]) to directly compare our CNN to CNNs of varying depth, while EfficientNet-B0 (Tan and Le [2019]) was specifically chosen to assess if its highly optimised architecture could beat our model in latency.

To establish performance baselines, the selected models were adapted for our specific tasks. For hit frame identification, the final classification layer was replaced with a single linear neuron to regress the hit likelihood score. For landing point prediction, the input layer was modified to accept stacked frame sequences, and the output layer was replaced with a two-neuron head using a Sigmoid activation to predict normalised (X, Y) coordinates.

The performance of these pre-trained models against our custom CNNs is detailed in Sections 6.4 and 6.5.

6 Numerical Evaluation

6.1 Hit Frame Prediction Performance (CNN1)

The training data for CNN1 can be found in Figure 5. CNN1 was trained for a max of 150 epochs, with early stopping finishing training at Epoch 139 and yielding a final train and val loss of 0.0158 and 0.022 respectively. We can see that the Validation MAE graph is very noisy, and this is likely due to a small batch size. This was unavoidable due to computational constraints - the final models had to be trained locally on an Nvidia RTX 4060 GPU to avoid Google Colab file bottlenecks, and this GPU didn't allow a larger batch size. An easy improvement to this project would be running the training process in cloud to access a more powerful GPU.

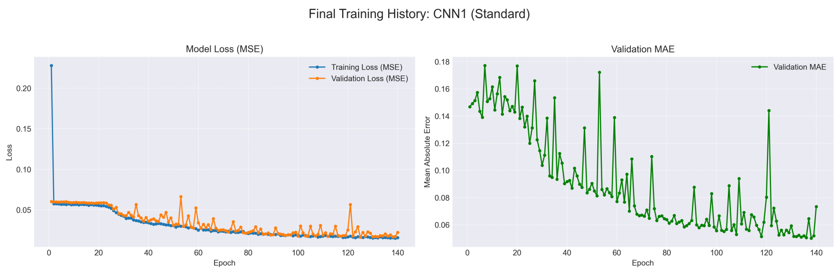


Figure 5: CNN1 Final Training Plot (Standard Training)

An example result for CNN1 is demonstrated in Figures 6 and 7. A directory of frames from an unseen video is passed into CNN1 sequentially. Figure 6 shows the predicted likelihood of each frame being the frame in which the ball was hit. Figure 7 shows the three frames identified as being the most likely frames in which the hit occurred. By observation, we can see that the model was correct in identifying the exact frame in which the ball was hit in this video, with `frame_0105.jpg` showing the tennis ball actually in contact with the racket.

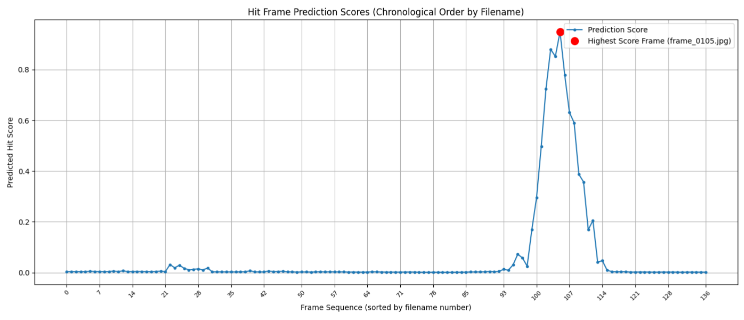
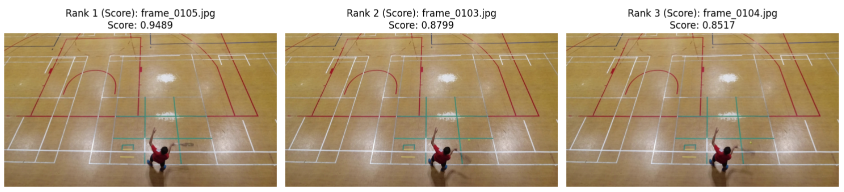


Figure 6: CNN1 Final Training Plot (Standard Training)


Rank 1 (Score): frame_0105.jpg
Score: 0.9489
Rank 2 (Score): frame_0103.jpg
Score: 0.8799
Rank 3 (Score): frame_0104.jpg
Score: 0.8517
Figure 7: CNN1 Final Training Plot (Standard Training)

6.2 Landing Point Prediction Performance (CNN2)

From Figure 8 we can observe an undesirable fact of our dataset - while we have many frames usable for CNN1 (in the order of 100,000), we have a limited number of videos ($\sim$240). This means that the training process will inherently be very noisy as we are forced to use a low batch size due to data availability - and this is observable from the Model Loss graph in Figure 8. When run for a maximum of 150 epochs, we found that it stopped after 63 epochs, with a final train loss of 0.0444 and val loss of 0.0459.

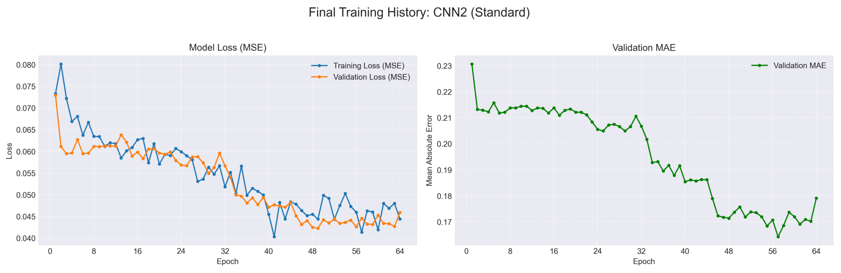


Figure 8: CNN2 Final Training Plot (Standard Training)

An example output of CNN2 can be found visualised in Figure 10 shown in Section 6.5.

6.3 Joint Training Performance (CNN1 + CNN2)

From Figure 9 we can observe a few things. Firstly, the joint implementation finishes training in 35 epochs - lower than either of the other CNNs individually. This is likely due to the penalty term dominating the other components of the loss - an issue we tried to address by making the penalty weighting parameter, λ, decay in proportion to the overall loss - though this problem would be more effectively mitigated by the approach outlined in Section 7. CNN2 reaches a final val loss of 0.0609, higher than on the individual train (0.0459). Similarly CNN1 reaches a final train loss of 0.0477 in the joint train and 0.0158 in the individual train - a significant decrease. We can also see in the middle graph (red line) that the penalty term (representing the distance of hit frame prediction from the true frame as opposed to raw weight difference) fluctuated without decreasing - we imagine this to mean that the penalty term keeps the top prediction frame from getting too far from the true frame, but the CNN1 is not incentivised to minimise past a certain point as this degrades prediction accuracy from CNN2. We can also observe very noisy training in CNN2, as well as some noticeable overfitting in the uptick of the right-most graph. We believe these issues would be heavily mitigated by the potential improvements discussed in Section 7.

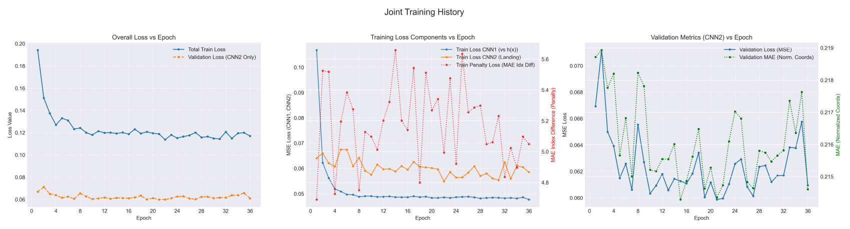


Figure 9: Joint Final Training Plot

6.4 Pre-Trained Performance

When comparing the performance of our custom architectures against pre-trained models fine-tuned for this purpose, we observe positive results (see Tables 1 and 2). CNN1 performs better than 2 out of 3 of the pre-trained models tested for that task, while CNN2 performs better than all 3. We also find that our light-weight custom models have much lower latency (see Table 3) compared to the pre-trained models, likely due to our models being more optimised for these specific tasks.

6.5 Combined Pipeline Performance (CNN1 + CNN2)

The success of our prediction system comes down to its performance when the two CNNs are working end-to-end: a video is being played for which CNN1 identifies the frame in which the tennis shot is

Model	Validation MSE Loss
ResNet-50-HF	0.016437
ViT-Base-Patch16-224	0.058209
ConvNeXt-Tiny-HF	0.058203
Custom CNN1	0.0222

Table 1: Pre-trained Hit Frame Results

Model	Val MSE Loss
ResNet-18-timm	0.128865
ResNet-34-timm	0.111751
efficientnet_b0-timm	0.046955
Custom CNN2	0.0459

Table 2: Pre-trained Ball Landing Results

taken, and then CNN2 predicts the landing spot of that shot before it even happens. To evaluate the performance of our combined pipeline, we compare it with a similar pipeline made up of two of the pre-trained models (ResNet-50-HF for CNN1 and efficientnet_b0-timm for CNN2).

Table 3: Comparison of Detected Hit Frame Results (Own vs. Pre-trained Models)

Model Type	Frame ID	Hit Score	Hit Frame Latency (ms)	Landing Coords (X,Y)	Landing Predict Latency (ms)
Own Model	71	0.6580	32	(0.51, 0.37)	182
	72	0.5254	35	(0.51, 0.37)	195
Pre-trained	70	0.5226	322	(0.535, 0.282)	1137
	71	0.7858	362	(0.537, 0.274)	934
	72	0.9247	312	(0.535, 0.272)	1097
	73	0.6899	297	(0.531, 0.273)	929

Table 3 shows a direct comparison of the performance outputs for key frames between our model and the fine-tuned pre-trained models. A contrast that is immediately apparent is in processing latency. The custom model exhibits significantly lower latency for both hit detection (∼32-35ms) and landing prediction (<200ms) compared to the pre-trained model, which required approximately 300-400ms for hit detection and 900-1100ms for landing prediction - roughly an order of magnitude slower for hit detection and 5 times slower for landing prediction. Furthermore, the models differed in their hit identification; the pre-trained model flagged a wider range of frames (70-73) compared to the custom model (71-72) showing our model did better at being selective, and not wasting compute time running many predictions from many similar frames, unlike the pretrained. While the pre-trained model shows potential sensitivity across more frames, its substantially higher latency severely impacts its feasibility for real-time analysis compared to our much faster custom architecture.

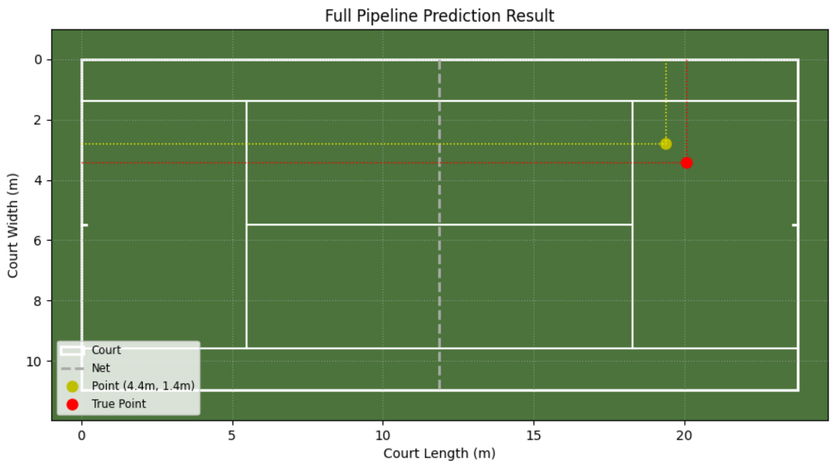


Figure 10: Full pipeline prediction of landing point, compared with true point.

The combined pipeline allows a live video to be played, the hit frames to be identified, and a landing point to be outputted for each hit frame. A visualisation of this output is shown in Figure 10. In this figure we see that the model's output (yellow) is markedly close to the true landing point (red). If, in practice, this pipeline was used to predict whether a ball would land in or out of the court, in this instance, it would have been successful.

7 Discussion

Despite yielding promising results, our project was bound by several key limitations:

Data Our dataset is tennis shots taken exclusively in two settings and from a single camera angle. While augmentations helped to address this, an important limitation to our project was our access to more varied data. The acquisition of better data, however, is not a trivial task. An expansion of our data to generalise better on professional tennis video streams would require large amounts of data from copyrighted sources with landing coordinates attached. We imagine this process of labeling the data would involve using a separate motion tracking neural network to track the trajectory of the ball, and then map the trajectory on the image to a trajectory on a court - possibly using a linear algebraic projection. We believe that this is the single most valuable development of this project, but could lead to significant computational requirements and legal considerations.

Joint Training Development While our Joint Training implementation yielded interesting results, it left a lot to be desired. Currently, the loss function that both CNNs are updated on is a linear combination of the loss of CNN1, the loss of CNN2, and a penalty term to discourage predicting too late in the shot. Ideally, the loss function would consist of only CNN2 loss and the penalty term, with weight updates passed backwards through CNN2 and backwards through CNN1 in one contiguous update. This would represent a truly end-to-end model where CNN1 learns directly from the output of CNN2, rather than from a more abstracted loss function which is the current configuration. A further change could be to fully integrate the complex frame weight assignment function into the training process. This would mean that the output loss from CNN2 would directly contribute to the parameters of the weighting function and therefore the curvature of the decay of the frame weights. This is analogous to the combined neural net being able to take a view on the importance of frames surrounding the hit frame - a truly contiguous tennis shot prediction system - but would require an entirely custom weight update method outside of the standard PyTorch methods.

Performance A critical aspect for our intended application is latency. Our pipeline demonstrated significantly lower inference times compared to the heavier pre-trained models (Table 3) on our test hardware (NVIDAI RTX 4060). Given that a tennis shot's flight time is often cited as $\sim$450ms or more, this leaves a potential buffer of over 250ms. This suggests the system possesses the potential for near real-time application, allowing just enough time for interpretation and potential action (e.g., placing a bet) before the outcome is known. However, this buffer could still be increased. Firstly, applying static quantisation to convert the model weights to INT8 may provide substantial speedups with minimal accuracy degradation by reducing memory bandwidth requirements and enabling faster computations (PyTorch Contributors [2025]). Secondly, compiling the model with ONNX runtime could further accelerate inference (ONNX Contributors [2025]).

8 Conclusion

We developed a dual-CNN architecture for predicting the landing coordinates of tennis shots before the outcome has been realised, specifically targeting real-time decision-making scenarios such as sports betting. Our results show the system described achieves sufficient latency and accuracy to potentially be successful and provide value in this scenario. However, key limitations include a lack of professional tennis video data, leading to poor generalisation on varied live feeds. Further exploration in this field is limited by the acquisition of large quantities of wide-ranging labelled data, as well as the infrastructure to support training on a greatly expanded dataset.

Statement about individual contributions

The project work was distributed among the group members as follows:

- Member 1 (Candidate Number: 50537): Contribution Percentage: 25%
  - Summary of contribution: Keystone adjustment for data augmentation. Rewriting of some code to enable Object Oriented Programming. Experimentation/ hyperparameter tuning for both CNNs. Coding of early stopping functionality. Write-up for data augmentation and experimentation. General help with writing/ editing of report. Creation of slides. Also, train/ val/ test split.
- Member 2 (Candidate Number: 44158): Contribution Percentage: 24.5%
  - Summary of contribution: Conducted research into alternative methods and existing work. Constantly collaborated with other group members to support with model development and to ensure clear and accurate documentation and running of experiments and results. Set up report, and conducted proofreads to ensure full compliance with NeurIPS standards. Finalised report structure and presentation, ensuring consistency in formatting, citation style, figures and tables.
- Member 3 (Candidate Number: 39706): Contribution Percentage: 25%
  - Summary of contribution: Selection of pre-trained models and developing fine tuning code. Development of live model prediction UI. Write-up for pre-trained model and full pipeline. Editing of Slides. Writing for Introduction and Related Work.
- Member 4 (Candidate Number: 50615): Contribution Percentage: 25.5%
  - Summary of contribution: Developed separate and joint training mechanisms and ran and supervised final train. Wrote code-base allowing final training outside of .ipynb to minimise CPU bottleneck. Wrote Methodology and Discussion section and edited report.

We confirm that these contributions are accurately represented.

References

Aniket Chatterjee, Priyansh Bhargava, and Vivek Patel. Tennis shot recognition and classification using deep learning. `https://cs230.stanford.edu/projects_winter_2020/reports/32209028.pdf`, 2020. Stanford CS230 Course Project, Winter 2020.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations (ICLR)*, 2021. URL `https://arxiv.org/abs/2010.11929`. Basis for google/vit-base-patch16-224 model.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. URL `https://arxiv.org/abs/1512.03385`. Basis for ResNet architecture (microsoft/resnet-50, timm ResNet-18/34).

Hugging Face. Hugging face - the ai community building the future. `https://huggingface.co/`, 2024. Accessed: 2025-05-04.

Kalin Guanlun Lai, Hsu-Chun Huang, Wei-Ting Lin, Shang-Yi Lin, and Kawuu Weicheng Lin. Tennis shot side-view and top-view data set for player analysis in tennist. *Data in Brief*, 54:110438, 2024. ISSN 2352-3409. doi: 10.1016/j.dib.2024.110438. URL `https://www.sciencedirect.com/science/article/pii/S2352340924004074`.

Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11976–11986, 2022. URL `https://arxiv.org/abs/2201.03545`. Basis for ConvNeXt architecture (facebook/convnext-tiny-224).

ONNX Contributors. Onnx. `https://onnx.ai/`, 2025.

PyTorch Contributors. Quantization documentation. `https://pytorch.org/docs/stable/quantization.html`, 2025. Accessed: 2025-05-02.

Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 6105–6114, 2019. URL `https://arxiv.org/abs/1905.11946`. Basis for EfficientNet architecture (timm efficientnet$_b0$).

Kirill Yastrebov. Tracknet: Tracking tennis ball using deep learning. `https://github.com/yastrebksv/TrackNet`, 2019. Accessed: 2025-04-27.