

Investigating the Effect of Temporal Sequence Length and Additive Attention Pooling on BiLSTM-Based Fencing Outcome Recognition

Qihong Wu, Yang Zhang

Shanghai Qingpu World Foreign Language High School, Shanghai, China

Email: alexwu080928@gmail.com

How to cite this paper: Wu, Q.H. and Zhang, Y. (2026) Investigating the Effect of Temporal Sequence Length and Additive Attention Pooling on BiLSTM-Based Fencing Outcome Recognition. *Journal of Computer and Communications*, 14, 20-54. <https://doi.org/10.4236/jcc.2026.1410002>

Received: September 3, 2026

Accepted: October 19, 2026

Published: October 22, 2026

Copyright © 2026 by author(s) and Scientific Research Publishing Inc. This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).

<http://creativecommons.org/licenses/by/4.0/>



Open Access

Abstract

We examine the impacts of temporal sequence length and additive temporal attention pooling on the automated classification performance of a Bidirectional Long Short-Term Memory (BiLSTM) model for fencing outcome recognition. Human pose data was modelled via the OpenPose BODY_25 key-points, and the fencing outcomes were then labeled in each exchange (*i.e.*, left-fencer score, right-fencer score, no valid score). A controlled 3×2 experimental design was employed with varying sequence lengths (25, 50, or 75 frames) with or without additive temporal attention. All six setups were evaluated on a single fixed source-disjoint training, validation and test split with five matched training seeds for each setup. Only the best validation accuracy was used for model selection. Our best performing model was an attention model with a sequence length of 25 frames and a mean best validation accuracy of 58.93%. Nevertheless, its 0.18-percentage-point improvement on the 25-frame no-attention model was both statistically and practically insignificant. For no-attention, increasing the sequence length did not yield meaningful performance gain. We see that introducing attention did not make much of a difference when we used 25 frames, but improved the mean best validation accuracy by 1.77 percentage points when we used 50 frames, and 3.23 percentage points when we used 75 frames, when compared to the no-attention version of the respective model. The test accuracy of the model, which was chosen at the best validation performance, was 57.35%, with a macro-F1 of 48.29% and a balanced accuracy of 51.14%. Performance on the underrepresented class of no valid score remained poor. This study suggests that we may not benefit from longer temporal windowing for fencing outcome recognition and additive temporal attention may become counterproductive at longer sequencing. The proposed classifier, hence, holds its value as an experimental model but is not yet effective as a refereeing tool.

Keywords

Fencing Outcome Recognition, Human Pose Estimation, OpenPose, BiLSTM, Temporal Sequence Length, Additive Temporal Attention, Imbalanced Classification

1. Introduction

Computer vision has begun to be applied automatically to analyse sports video for performance reporting and as an aid to judging. A difficult application is to fencing, where an exchange can unfold in a few seconds: two fencers are moving at the same time and who scored depends on a temporal chain of movements to prepare attack, attack and counter rather than a single pose. A pose-based classification system is far from a substitute for the proper fencing judging systems, but we can nonetheless ask whether one can be built that simply uses body movement to discriminate a left-fencer score, a right-fencer score and an exchange with no valid score.

This article considers this question using a small video-processing pipeline. First a pose estimation model, OpenPose, is used to represent each frame as the coordinates for 25 anatomical keypoints for each fencer, stripping out background and colour information while preserving a description of the motion with an organised structure [1]. The resulting time series is then processed by a Bidirectional Long Short-Term Memory network (BiLSTM), which reads the full sequence once forwards and once backwards, before outputting its classification of the exchange [2] [3].

Two controllable characteristics of this pipeline will be the subject of this exploration. The first is length of the temporal sequence: longer sequences have the potential to maintain the development and final outcome of an interactive exchange, but are more computationally expensive and potentially contain less helpful initial movement. The second is the additive temporal attention pooling, which allows for weighted contributions of each frame. Attention may provide benefit when a few critical moments dominate a longer sequence, but evidence may be evenly split across the length of an action and attention unhelpful in that case. Testing both characteristics in combination therefore allows for the discovery not only of their independent impacts, but also if the usefulness of attention is modulated by sequence length.

To test these decisions in a controlled way, each of the three input lengths (25, 50 and 75 frames), are tested with and without the attention pooling layer with five matched sets of random seeds. The six configurations are compared with their validation accuracy; architecture selection is based only on validation performance, while test results are reported for the validation-selected architecture. The research question is: **To what extent do temporal sequence length and additive temporal attention pooling influence the classification accuracy of a Bidirectional LSTM model in automated fencing match outcome recognition?**

2. Key Terms

The following terms are defined before the background information and methodology explanation so terms could be used consistently throughout the essay.

- **Temporal sequence length (T):** The number of continuous pose frames given to the model. In this experiment, the values are 25, 50 and 75.
- **Keypoint:** A two-dimensional coordinate locating an anatomical landmark. BODY_25 provides 25 indexed landmarks for each detected person.
- **BiLSTM:** A bidirectional recurrent neural network where a sequence of input is sent forwards and backwards through the network. The results are then merged to generate the final output.
- **Additive temporal attention pooling:** An attentive (learned) scalar scoring function over BiLSTM time steps, which is normalized and then used in a weighted sum. Not equal to Transformer self-attention.
- **Random seed:** Initialization seed used for pseudorandom number generating operations (initialization of parameters, dropout masks, batch order, etc.). Same seed for repeated experiments ensures same initial state.
- **Training set:** Data used to update model parameters during optimisation.
- **Validation set:** Held-out development data used to select checkpoints and compare configurations without consulting test labels.
- **Test set:** Source-disjoint data used to estimate generalisation; test performance was not used for architecture selection.
- **Macro-F1:** The unweighted mean of the three class-specific F1 scores, so the rare class contributes equally to the result.
- **Balanced accuracy:** The arithmetic mean of recall over the three classes.

3. Background

3.1. Human Pose Estimation

Human pose estimation is defined as the task of predicting the location of anatomical landmarks from visual inputs. OpenPose is a human pose estimation algorithm that identifies each human in real time with a method called Part Affinity Fields [1]. BODY_25 is one such representation of each human as 25 joints indexed by numbers with bones between them, as in **Figure 1**. The idea behind this type of representation is to find a lower dimensioned feature that preserves as much of the body's articulation, while decreasing sensitivity to factors like clothing, lighting and background. This very abstraction, however, also strips away some potentially dispositive details, such as a weapon-tip position, a blade in contact with something or scoring-box lights, an issue which we will consider later.

3.2. RNNs, LSTMs and Bidirectionality

In an RNN (recurrent neural network), a sequence of inputs is passed through and a hidden state is updated at every time-step. This allows a previous frame to affect how a later frame is interpreted. Recurrence thus better suits speech, lan-

guage and action sequences where input order matters than a feed-forward model [4]. However, gradients can be shrunk by repeated multiplication during back-propagation, making a standard RNN unreliable at learning dependencies across many time-steps.

LSTM (Long Short-Term Memory) networks can solve this issue with gated memory paths. Three gates—the input, forget and output gates—govern what is allowed to pass into memory, what is to be retained, and what is made available to the next layer. This grants information a more stable trajectory through time [2]. The LSTM thus matches sequences of 25 - 75 frames better in theory than a standard RNN, as indicated in **Figure 2**.

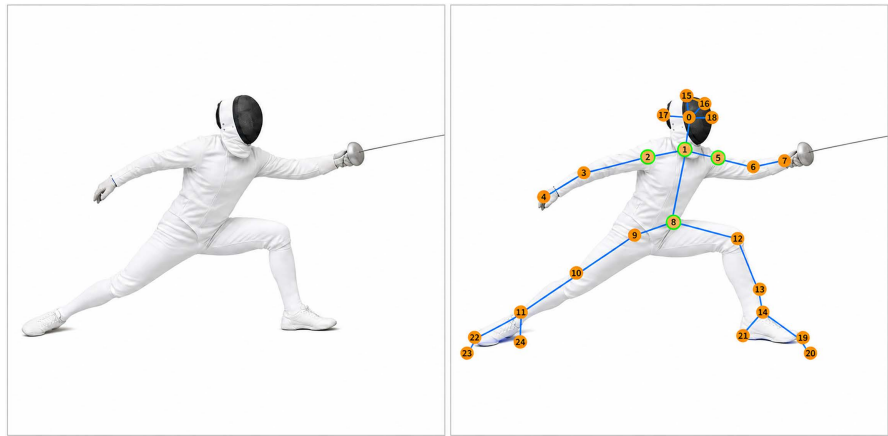
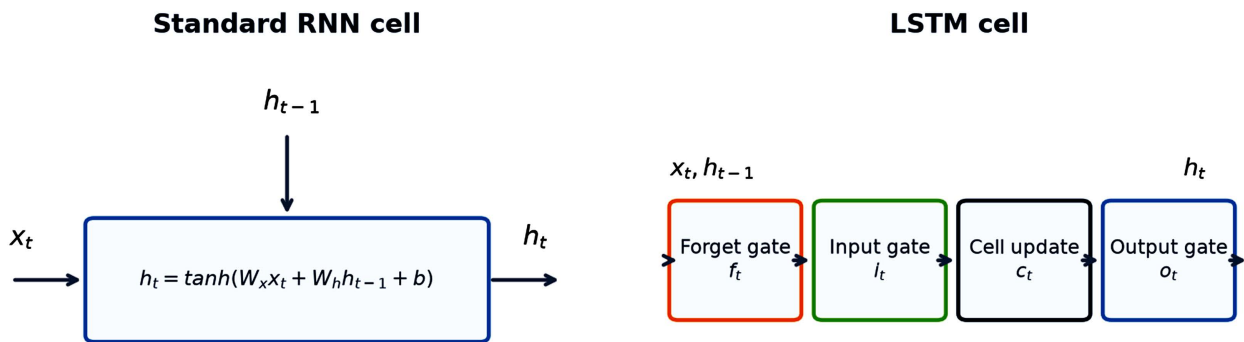


Figure 1. OpenPose BODY_25 skeleton: unlabeled (left) and indexed (right).



Gates create an explicit memory path for longer dependencies.

Figure 2. Standard RNN update and the gated memory structure of an LSTM.

A bidirectional LSTM simply places a forward and a backward LSTM together to allow both representations to draw on the context from either side of a given time step [3]. This seems well suited to the task at hand, since we can fully sample both the exchange and resolution before performing a classification: preparation can give us clues about what the eventual strike will be, and the resolution can provide context for an earlier strike. The bidirectional LSTMs would not be well suited for a strictly online prediction, where subsequent frames have not yet been recorded.

3.3. Additive Temporal Attention

Attention was introduced as an approach to prevent an entire sequence being condensed into a single encoder state [5]. Within this study a compact feed-forward network is used to give each 512-dimensional BiLSTM output h_t an unnormalised relevance score e_t . These are then transformed via a softmax function to produce attention weights α_t which are now normalised to one. The attention weights α_t are then applied as a weighted sum of outputs to create a context vector c :

$$e_t = w_2^T \tanh(W_1 h_t + b_1) + b_2$$

$$\alpha_t = \frac{\exp(e_t)}{\sum_{k=1}^T \exp(e_k)}, \quad c = \sum_{t=1}^T \alpha_t h_t$$

where h_t is the hidden state of the bidirectional LSTM from frame t , e_t is the unnormalised score, α_t is the temporal attention weight, and c is the attention-pooled representation passed to the classification head. This is an experiment in seeing if you get any recognition boost by considering the weights to be not equal for the individual frames. It is temporal attention pooling as opposed to the self-attention mechanism in Transformer because the frames do not communicate with one another through query-key pairs.

4. Methodology

Experiment setup, where **Figure 3** is adopted: The same preprocessed videos are assigned to the source-disjoint datasets, the two independent factors determine six controllable settings, the validation results determine the architecture to use, and the main text reports test performance for the validation-selected architecture.

Experimental pipeline

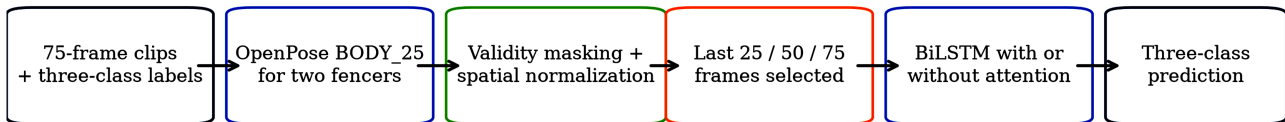


Figure 3. Experimental design and evaluation workflow.

4.1. Dataset and Labels

The published Sport Fencing Matches dataset contains AI, OpenPose, descriptor, and source URL information for publicly available fencing videos [6]. A preprocessing yielded 9370 clips: as 4846 structurally valid clips, and 4524 repaired clips. Each model input was derived from an array of (75, 2, 25, 2), *i.e.*, a sequence of frames with 2 fencers and 25 BODY_25 landmarks and their x/y coordinates. The two pose streams were ordered by horizontal position, with the fencer having the smaller x-coordinate assigned to the visual-left stream and the other to the visual-

right stream. During preprocessing, we used detection confidence to determine valid joints, but this was not fed to the model inputs. A source frame and its corresponding pose representation are linked in **Figure 4**. **Figure 5** shows that the coordinate-only information retain the posture of the body of the two fencers.

Label assignment was straight forward: we used 1-0-0 as class 0 (left fencer scored), 0-0-1 as class 1 (right fencer scored), and 0-1-0 as class 2 (no valid score). A prefix of perfect_ and fixed_ was stripped before concatenating identifiers to the decriptor table. There are 4337, 4501, and 532 number of class-0, -1, and -2 clips from explicit checks; thus, class 2 accounted for only 5.68%.

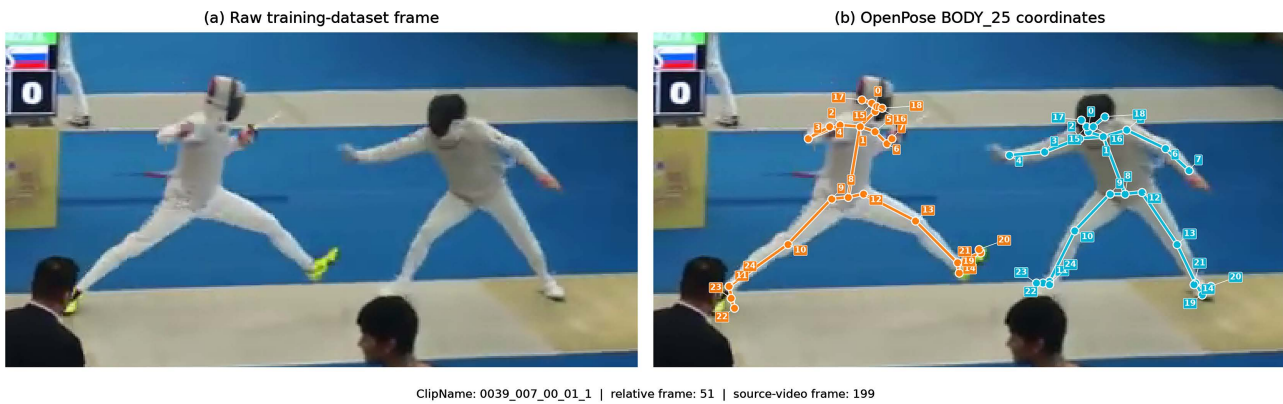


Figure 4. Original fencing frame and corresponding BODY_25 landmarks.

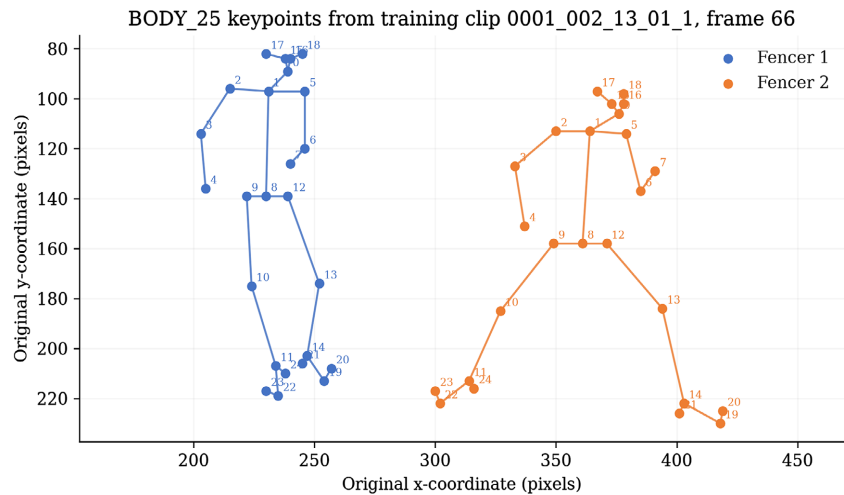


Figure 5. Coordinate-only reconstruction used to verify preprocessing.

4.2. Independent Variables and Experimental Conditions

Two experimental factors are the length of temporal sequence (25, 50 or 75 frames) and the presence/absence of additive temporal attention pooling, for a total of six experimental conditions, as listed in **Table 1**. All source video clips have 75 frames. For 50 and 25 inputs, the later two-thirds and one-third of the full sequence are sliced, respectively, with the effect that each of the shorter sequences

is an embedded subset of each of the longer sequences. This method allows us to isolate and test the significance of additional early context while avoiding alterations in the rest of the frame sequence. The three frame lengths are experimental and relative to the annotated exchange itself; they are not universal optima for fencing. The start and end boundaries were inherited from the pre-segmented clips in the source dataset rather than manually re-annotated in this study. A similar treatment of temporal scope as a design parameter can be found in action-recognition methods which sample video data over time [7].

Table 1. Six experimental conditions defined by the two independent variables.

Configuration	Sequence length (frames)	Additive attention
C1	25	No
C2	25	Yes
C3	50	No
C4	50	Yes
C5	75	No
C6	75	Yes

4.3. Preprocessing and Notation

The five operations below form an ordered pipeline, summarized in **Figure 6**. Each stage was applied identically in all six configurations so that preprocessing did not become an uncontrolled variable.



Figure 6. Ordered preprocessing pipeline applied to every clip.

Frame fixing. Fencers with clips with less than 75 frames, repeated their last good frame, rather than having a sudden ending of all zeros. A 25×2 zero matrix signified an invalid fencer.

Validity masking. Joint validity was indicated when its confidence of being detected was above zero, and its position was not $[0, 0]$. Undetected joint positions remained $[0, 0]$ following normalisation rather than being projected to any arbitrary far location.

Spatial normalisation. For frame t and fencer f the centre $c_{t,f}$ was set as the mid hip keypoint when valid, the mid point between the two shoulders otherwise and the last valid centre otherwise. The scale $s_{t,f}$ was set as the distance between the mid-hip and the neck keypoint when at least one pixel, and then the distance between the shoulders when valid, and the last valid scale otherwise. A fencer with no valid centre or scale had its positions masked with zeros.

$$\hat{p}_{t,f,j} = \frac{p_{t,f,j} - c_{t,f}}{s_{t,f}}$$

In the above formula, $p_{t,f,j}$ denotes the untransformed two-dimensional location of joint j , and $\hat{p}_{t,f,j}$ its normalised location. Subtracting $c_{t,f}$ leads to less sensitivity to image position; dividing by $s_{t,f}$ leads to less sensitivity to apparent body scale. Joint coordinates that were not valid were not transformed.

Temporal slicing. Identical 75-frame array was applied across all conditions. The model was given the last T frames, where T equals to 25, 50 or 75. The end of the annotated exchange was thus kept the same across conditions.

Feature reshaping and BatchNorm. The $2 \times 25 \times 2$ spatial locations per frame were flattened to 100 features. We then applied batch norm on the feature dimension before passing the features to the BiLSTM:

$$BN(x_k) = \frac{\gamma_k (x_k - \mu_{B,k})}{\sqrt{\sigma_{B,k}^2 + \epsilon}} + \beta_k$$

where $\mu_{B,k}$, $\sigma_{B,k}^2$ denotes the mean and variance over the mini-batch for the feature k , γ_k , β_k are parameters with the same dimensions as the feature and epsilon is added for avoiding division-by-zero. Batch normalisation was applied in order to stabilise the training process by normalizing the layer's input during training [8].

4.4. Group-Disjoint Train, Validation and Test Sets

If we were to split the individual videos at random then sections of videos from the same Youtube match may end up in the training set and the test set, leading to an optimistic bias. Hence, we have considered the sources (URLs) as groups. A 7-fold stratified group cross-validation, then assigned 5 for training, and 1 for validation and one for testing preserving roughly the proportion of each class in the group. The resulting data had a total of 6683 clips belonging to 948 source videos, 1344 clips belonging to 192 sources and 1343 clips belonging to 192 sources as illustrated in **Figure 7**, with no URL present in more than one of the set.

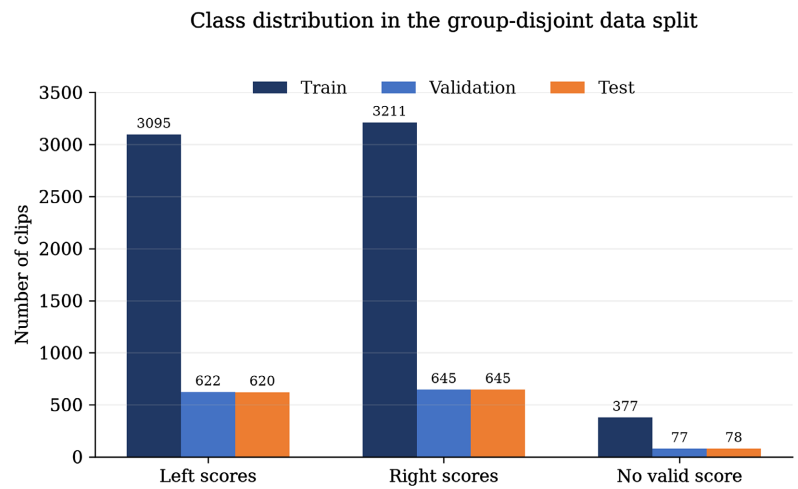


Figure 7. Class counts in the source-disjoint training, validation and test sets.

4.5. Model Architectures and Design Justification

Figure 8, both models used batch normalization of input layer, a pair of BiLSTM layers with 256 units in each (hidden) direction and 0.4 dropout between each recurrent layer, a 512-to-128-to-3 neuron classifier with batch normalization, ReLU and 0.5 dropout. We used LSTM because motion sequences are inherently sequential, and sometimes need to depend on several previous frames (up to 75), and bidirectional because the entire clip is available when it is time to make a classification decision. For all three conditions we used 256 units in order to avoid having model width be a third independent variate.

The model without attention just took the final forward and backward state and concatenated them as the 512-dimensional context. The attention model scored all 512-dimensional outputs for each time step with a 512-to-128-to-1 additive network and then took their weighted sum. Both models had 2,376,651 and 2,442,444 trainable parameters respectively; we will state the extra 65,793 as a potential difference in model complexity, though the training outcome does not support the simple possibility that one is overfitting more than the other.

Controlled comparison of the two model architectures

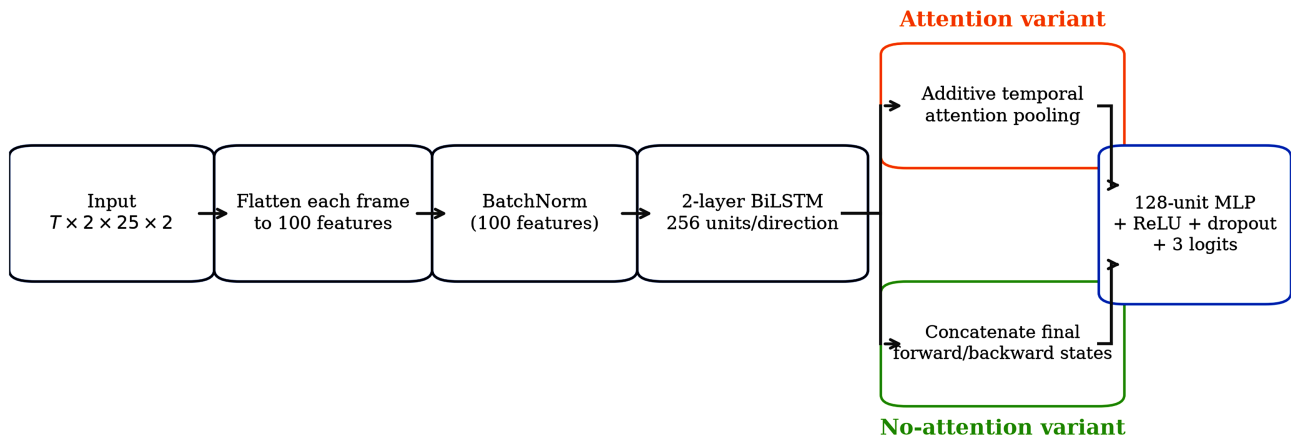


Figure 8. Shared BiLSTM backbone and the single controlled architectural difference.

4.6. Training Controls, Loss and Checkpoint Selection

During training, a batch size of 32 was used with 20 epochs, an L2 weight decay of 1×10^{-4} , and the Adam optimizer with a learning rate of 0.001 for all models. The Adam optimizer utilizes estimates of the first- and second-order moments to perform adaptive updates [9] and dropout is used to prevent co-adaptation among units [10]. Instead of being specifically tuned to each condition, we fixed these hyperparameters for all conditions to maintain the controlled comparison.

Because one of the classes was under-represented, we mitigated the issue by applying inverse-frequency weighting of the loss using a cross-entropy loss. The class weights were [0.720, 0.694, 5.909] given a training class count of [3095, 3211, 377], which means errors in the no-valid-score class weighed more heavily in the loss.

$$L = -w_y \log(p_y)$$

Each run had five random-number generators for Python, NumPy, PyTorch and CUDA; (42, 52, 62, 72 and 82) that were used to match the parameters initialisation, the dropout masks and the data-loader shuffling. In order to allow reproducible operations we asked for deterministic algorithms to be used and we turned off the cuDNN benchmarking. Using the same random-number generator for different conditions should reduce the randomness caused by the deep-learning process. Repeating the conditions with the same random number seeds will account for some variability that may occur when using deep learning, even when one method seems reproducible [11].

The controlled training settings used across all configurations are summarized in **Table 2**.

Table 2. Controlled variables and training configuration.

Controlled element	Setting used in every configuration
Dataset split	Same training set/validation set/testing set
Training schedule	20 epochs; batch size 32; Adam; learning rate 0.001
Regularization	LSTM dropout 0.4; classifier dropout 0.5; L2 weight decay 0.0001
Loss	Inverse-frequency weighted cross-entropy; weights 0.720, 0.694, 5.909
Randomness	Seeds 42, 52, 62, 72 and 82 applied to Python, NumPy, PyTorch, CUDA, initialization, dropout and data shuffling
Evaluation	Architecture selection based only on validation accuracy; test metrics not used for model selection

For each run, a checkpoint corresponding to the run with maximum validation accuracy was recorded. Validation and not test data, so not an exposure to the test labels. Training and validation accuracy for each final epoch was also recorded, to observe convergence characteristics. When comparing results between configurations, reported is the mean and the sample standard deviation of five results on matching seeds, so single unusually good runs don't dictate the outcome.

4.7. Dependent Variables and Statistical Analysis

Overall, we choose best validation accuracy as the primary dependent variable: the maximum validation accuracy achieved during a given set of 20 training epochs for each configuration, averaged over the five matched seeds. We retain final epoch validation accuracy as a convergence measure, and final epoch training accuracy as a diagnostic of fit. The architecture selection is determined by validation, and not test, performance. In the main analysis, test-set results are reported for the validation-selected architecture using accuracy, macro-F1, balanced accuracy and class-wise precision/recall. Macro-F1 treats all classes equally while balanced accuracy gives equal weight to each class's recall, thus ensuring that the 5.68%

minority class does not get swallowed up in overall accuracy [12].

Since the same seeds were run for the paired conditions, we use exploratory paired t-tests comparing validation accuracy with and without attention at each sequence length as well as selected comparisons of sequence-length conditions. Given only five pairs and several comparisons, the resulting p-values are descriptive rather than a basis for model selection.

5. Results and Analysis

5.1. Aggregate Performance

At the top there is a tight group in **Figure 9** and **Table 3**. The highest mean best validation accuracy score, 58.93%, is for the 25-frame attention models and has a sample standard deviation of 0.49 percentage points. The 25-frame and 50-frame no-attention models both scored 0.18 points worse at 58.75%. The margin between first and second is thus much smaller than the difference between seeds and should not be taken as evidence that attention is in general better.

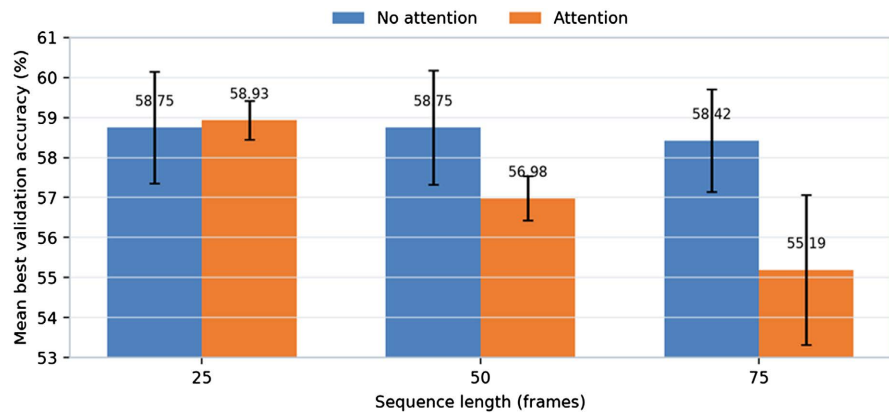


Figure 9. Mean best validation accuracy; error bars show one sample standard deviation.

Table 3. Mean (sample standard deviation) across five seeds; values are percentages.

Frames	Attention	Best validation acc. (%)	Final validation acc. (%)	Final training acc. (%)
25	No	58.75 (1.39)	57.87 (0.93)	75.77 (0.97)
25	Yes	58.93 (0.49)	57.54 (1.15)	70.66 (1.45)
50	No	58.75 (1.42)	57.10 (2.32)	72.96 (4.66)
50	Yes	56.98 (0.56)	55.67 (2.35)	65.08 (2.86)
75	No	58.42 (1.28)	56.76 (3.05)	71.58 (3.08)
75	Yes	55.19 (1.88)	54.81 (1.96)	66.14 (3.52)

5.2. Effect of Temporal Sequence Length

In the no-attention model, the average best validation accuracy did not differ between 25 and 50 frames (58.75%), and was very slightly decreased with 75 frames

(58.42%). A test for a significant difference between sequence lengths gave 50 frames - 25 frames = 0.00 points ($p = 1.000$), and 75 frames - 50 frames = -0.33 points ($p = 0.594$). Thus, the adjusted validation evidence does not provide any support for the initial hypothesis that longer sequences are systematically better.

A more conservative view is that 25 frames already include sufficient late-exchange evidence for this model and this dataset. Additional early frames might provide more helpful preparation context for a few video clips, but the overall advantage is outweighed by extra variability and only loosely relevant motion information. The impact of sequence length is thus at maximum a saturating small one with the conditions tested here.

5.3. Effect of Additive Temporal Attention

The within-seed changes in best validation accuracy after adding attention are shown in **Figure 10**.

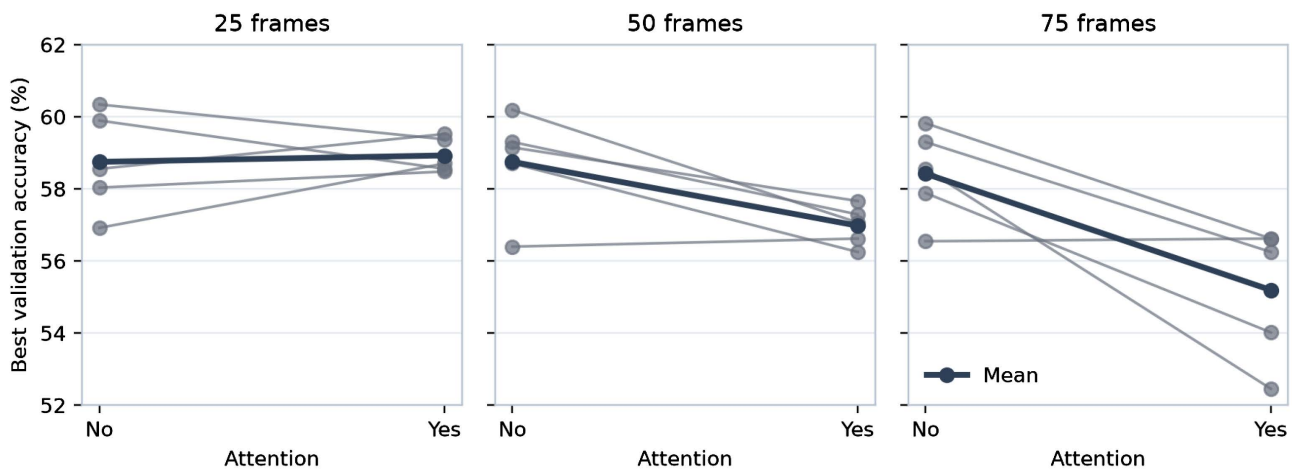


Figure 10. Within-seed change in best validation accuracy when attention is added.

As shown in **Table 4**, the descriptive pattern suggests that the effect of attention may differ by sequence length. For 25 frames, attention altered the mean validation accuracy by just $+0.18$ points and the paired differences were not consistent ($p = 0.776$). At 50 frames, attention caused the accuracy to drop by 1.77 points ($p = 0.035$); at 75 frames, the decrease was 3.23 points ($p = 0.031$). These p -values are exploratory and not adjusted for multiple comparisons, but the directionality is consistent between the longer two conditions.

One possible explanation for this pattern is a mismatch between the attention pooling operation and evidence structure in fencing. A valid fencing outcome is a process rather than any particular key frame: approach, balance transfer, extension, and response and resolution all can provide subtle but cumulative evidence. The no-attention BiLSTM pools its full sequence of evidence in the final forward and backward state. Attention, however, allows frames to compete for a fixed total weight, which in longer sequences may dilute and suppress collectively useful but individually subtle context.

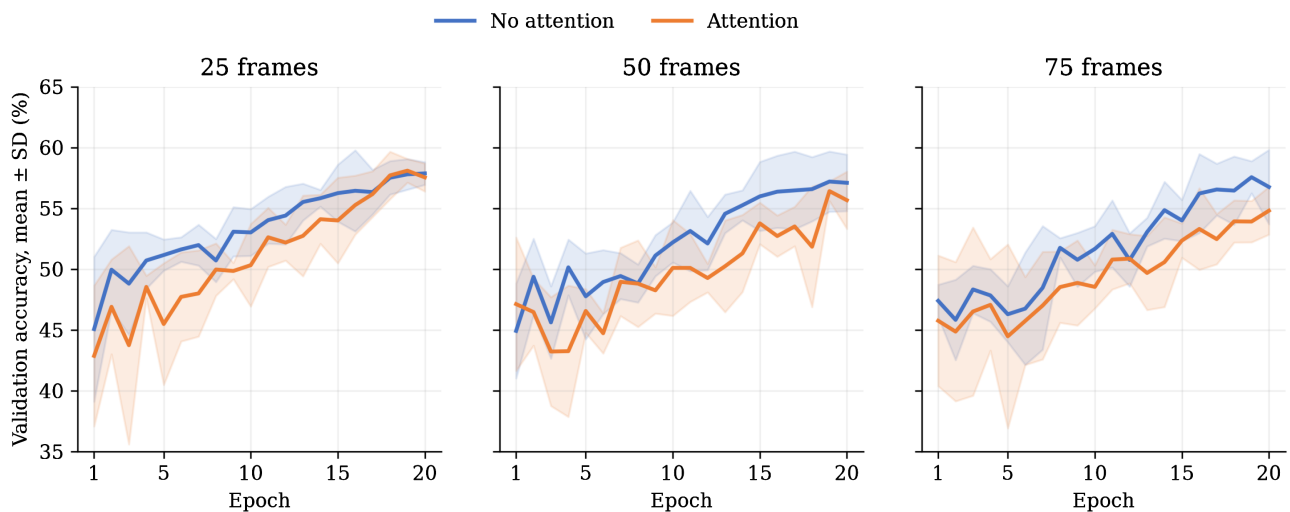
Table 4. Exploratory paired comparisons of best validation accuracy across matched seeds.

Comparison	Mean validation change	Paired t-test p
25 frames: attention - no attention	+0.18 pp	0.776
50 frames: attention - no attention	-1.77 pp	0.035
75 frames: attention - no attention	-3.23 pp	0.031
No attention: 50 - 25 frames	0.00 pp	1.000
No attention: 75 - 50 frames	-0.33 pp	0.594

The training evidence does not support a simple overfitting account. Attention introduced 65,793 additional parameters, but its final training accuracy was less than the corresponding no-attention model at every length (**Table 3**). If a model both fit the training data more strongly and generalised less well, this would be clearer evidence of overfitting. The observed pattern is more consistent with harder optimisation, or representational underfitting resulting from the pooling rule. This is only a hypothesis, however, as the attention weights were not saved for direct inspection.

5.4. Training Behaviour and Class-Level Error

Over 20 epochs, the validation accuracy for all cases improved with some fluctuations, as seen in **Figure 11**. The 50-frame and 75-frame attention conditions had consistently lower and more fluctuating validation accuracy. Checkpoint selection hence was done on the validation accuracy (as opposed to the last epoch). Each condition employed the same 20-epoch training schedule; hence this comparison is fair; however, separate tuning for the different architectures could result in different scores.

**Figure 11.** Mean validation-learning curves; shading shows one standard deviation.

5.5. Answer to the Research Question

Answer to the research question. Effect of temporal sequence length on validation accuracy (small and non-monotonic): no-attention conditions nearly tied between 25 and 75 frames. For additive temporal attention: the observed effect varied by sequence length, being negligible at 25 frames and negative at 50 and 75 frames. Under the pre-registered selection rule, the attention architecture with 25 frames was the top by mean best validation accuracy (58.93%), although its 0.18 point advantage over two of the no-attention alternatives was neither statistically nor practically significant.

The independent test performance of the validation-selected 25-frame attention model is summarized in **Table 5**.

Table 5. Independent test performance of the validation-selected 25-frame attention model.

Test metric	Mean (sample SD), %
Accuracy	57.35 (1.28)
Macro-F1	48.29 (1.99)
Balanced accuracy	51.14 (3.20)
Class 2 precision	17.89 (2.41)
Class 2 recall	36.15 (8.48)

According to the independent test, the choice of architecture has an accuracy better than the baseline of 48.04%, the majority class, but, as can be seen in **Figure 12**, it is not particularly reliable for the minority class of no valid score. For the five selected checkpoints, the average test accuracy was 57.35%, whereas the precision and recall for class 2 were 17.89% and 36.15%, respectively. This model can be considered as an experimental classifier, not ready for referee use.

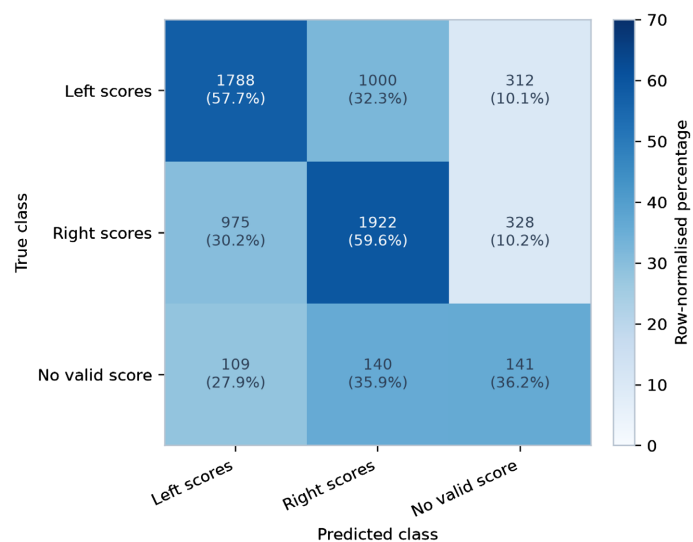


Figure 12. Aggregated test confusion for the validation-selected 25-frame attention model.

6. Limitations and Future Work

6.1. Pose-Only Evidence and Potential Label Leakage

Pose and body-track positions are available to the model; the motion of the weapon's tip, the hit of the blade on the target, the area and scoring-box lights are not. Thus, a visually interpretable motion might be not straightforwardly classifiable even after the official equipment emits an unambiguous verdict. The pose could be used together with the weapon and the target evidence identified independently for future applications. The scoring box lights should be eliminated from the classification window, or used as an external reference, since the visible verdict light would introduce a leakage in the labels, instead of testing the understanding of motion itself. More precise exchange-boundary annotation should therefore exclude frames that contain explicit post-outcome visual cues.

6.2. Minority-Class Scarcity and Augmentation

Among the total of 9370 video clips, only 532 were classified as no-valid-score instances. Inverse-frequency weighting successfully amplified this class's contribution to the loss, but failed to generate the null exchanges variety, double touches and off-target contacts that it lacked; low class-2 precision is the resulting effect. In the future, the collection should give special attention to this class, while subclasses of meaningful value are preserved. The application of conservative pose augmentation may introduce small noise to coordinates, temporal warping, translation or scale, so long as the labels for left and right remain intact and realistic fencing biomechanics are maintained.

6.3. Generalisability of the Source-Grouped Split

Although grouping images by their source URL ensures that videos from the same uploaded game do not end up in both test and development sets, this does not necessarily ensure independence by competition, by camera angle or by venue, due to missing information, or by athlete. The frozen model may therefore improve by exploiting similarities across related source domains. A better sequel would be to test the frozen model on a different recorded competition not included in the source dataset, recorded with different camera and athlete conditions.

6.4. Remaining Experimental Uncertainty

Seeds five seeds are more dependable than one seed, although they are fairly modest when it comes to statistical significance testing, and they use one split of the data set. We also froze hyperparameters to maintain the comparison on the IVs, so attention might not have seen its ideal learning rate, regularisation, or training rounds for each task on its own. The next step should involve pre-registering one main measure for each experiment, running more seeds and source-group partitions, hyperparameter tuning of the architecture within nested validation, and

holding out an external validation set for the final comparison. Although test metrics were computed for all configurations in the submitted implementation, architecture selection and all configuration comparisons in this study were based only on validation performance.

6.5. Confidence and Missing-Keypoint Representation

The validity masks were generated by the detection confidence and then dropped. This means that a zero coordinate might indicate that a joint is missing, or that it's a legitimate normalised co-ordinate at the origin, so that missing limbs can look like regular pose change when occluded or when the weapons move quickly. An additional input to the next model would be the confidence or explicit visibility mask on top of the coordinates. A graph based network might be able to preserve the BODY_25 connectivity rather than flattening both fencers to 100 independent features.

7. Conclusions

In this study, temporal sequence length and additive temporal attention were isolated in a controlled BiLSTM fencing classification experiments. All of the temporal sequence lengths were subarrays of the same 75-frame sequences; source video files were disjoint for training, validation and testing; six setups were repeated up to five times with matched seeds; and validation, not test, performance was used for comparing architectures.

For the validation set, not all longer lengths improved performance. In the attention-free setups, there was no more than a 0.33 percentage point difference for all three lengths. Attention was unremarkable for 25-frame input length, but decreased the mean of the best validation performance by 1.77 points at 50 frames and 3.23 points at 75 frames. These findings are consistent with the hypothesis that fencing movement classification relies on the accumulation of temporal information, and competitive attention-based pooling of frame-level evidence is detrimental in longer sequences.

According to the strict validation rule, 25-frame attention architecture should be used, with a mean best validated performance of 58.93%. The improvement of 0.18 points is too small to establish a good practical relationship. According to the results on an independent test set, the suggested 25-frame attention architecture obtained a 57.35% of mean (sample SD 1.28 points) test accuracy, 48.29% macro-F1, and 51.14% balanced accuracy. The findings are sufficient to answer the research question, but insufficient for practical deployment, as minority-class recognition is too poor, and any further advances will depend on richer evidence from the fencing weapons, more balanced training set and confidence-aware pose-related features.

Author Contributions

Qihong Wu carried out all stages of the research and wrote the manuscript. Yang

Zhang reviewed the final manuscript.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

- [1] Cao, Z., Simon, T., Wei, S. and Sheikh, Y. (2017). Realtime Multi-Person 2D Pose Estimation Using Part Affinity Fields. 2017 *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Honolulu, 21-26 July 2017, 7291-7299. <https://doi.org/10.1109/cvpr.2017.143>
- [2] Hochreiter, S. and Schmidhuber, J. (1997) Long Short-Term Memory. *Neural Computation*, **9**, 1735-1780. <https://doi.org/10.1162/neco.1997.9.8.1735>
- [3] Schuster, M. and Paliwal, K.K. (1997) Bidirectional Recurrent Neural Networks. *IEEE Transactions on Signal Processing*, **45**, 2673-2681. <https://doi.org/10.1109/78.650093>
- [4] Goodfellow, I. Bengio, Y. and Courville, A. (2016) Deep Learning. MIT Press.
- [5] Bahdanau, D., Cho, K. and Bengio, Y. (2015) Neural Machine Translation by Jointly Learning to Align and Translate. *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, San Diego, 7-9 May 2015, 1-2.
- [6] AlexPGD. Sport Fencing Matches AI. Kaggle. <https://www.kaggle.com/datasets/alexpqd/sport-fencing-matches-ai/data>
- [7] Wang, L., Xiong, Y., Wang, Z., Qiao, Y., Lin, D., Tang, X., *et al.* (2016) Temporal Segment Networks: Towards Good Practices for Deep Action Recognition. In: Leibe, B., *et al.*, Eds., *Computer Vision—ECCV 2016*, Springer International Publishing, 20-36. https://doi.org/10.1007/978-3-319-46484-8_2
- [8] Ioffe, S. and Szegedy, C. (2015) Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *Proceedings of the 32nd ICML*, Vol. 37, 448-456.
- [9] Kingma, D.P. and Ba, J. (2015) Adam: A Method for Stochastic Optimization. *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, San Diego, 7-9 May 2015, 2.
- [10] Srivastava, N., *et al.* (2014) Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, **15**, 1929-1958.
- [11] Bouthillier, X., Laurent, C. and Vincent, P. (2019) Unreproducible Research Is Reproducible. *Proceedings of the 36th ICML*, Vol. 97, 725-734.
- [12] Sokolova, M. and Lapalme, G. (2009) A Systematic Analysis of Performance Measures for Classification Tasks. *Information Processing & Management*, **45**, 427-437. <https://doi.org/10.1016/j.ipm.2009.03.002>

Appendix A. Reproducibility Record

This appendix is excluded from the main word count. It contains the complete submitted Python code and a 30-run summary. The experiment folder also contains aggregate_results.csv, per-epoch histories, classification reports, confusion matrices, selected checkpoints and the source-group split manifest.

Appendix A1. run_ee_experiments.py

```
import copy
import json
import os
import random
from pathlib import Path

import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.metrics import (
    accuracy_score,
    balanced_accuracy_score,
    classification_report,
    confusion_matrix,
    f1_score,
)
from sklearn.model_selection import (
    StratifiedGroupKFold,
)
from torch.utils.data import DataLoader, Subset

from dataset import FencingDataset
from modelv1 import FencingLSTM as AttentionModel
from modelv12 import FencingLSTM as NoAttentionModel

BASE_DIR = Path(
    r"F:\PycharmProjects\Fencing-Testing"
)
NPY_DIR = (
    BASE_DIR
    / "fencing_features_npy_experiment_v2"
)
LABEL_CSV = (
    BASE_DIR
    / "dataset"
    / "ClipDescriptorKaggle.csv"
)
OUTPUT_DIR = (
    BASE_DIR
    / "ee_experiment_results_v2"
)

SEQUENCE_LENGTHS = [25, 50, 75]
ATTENTION_OPTIONS = [False, True]
SEEDS = [42, 52, 62, 72, 82]

EPOCHS = 20
BATCH_SIZE = 32
LEARNING_RATE = 0.001
WEIGHT_DECAY = 1e-4

os.environ.setdefault(
    "CUBLAS_WORKSPACE_CONFIG",
    ":4096:8",
)

def set_seed(seed):
    os.environ["PYTHONHASHSEED"] = str(seed)

    random.seed(seed)
    np.random.seed(seed)

    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

    torch.use_deterministic_algorithms(
```

```
        True,
        warn_only=True,
    )

def create_split(dataset):
    splitter = StratifiedGroupKFold(
        n_splits=7,
        shuffle=True,
        random_state=42,
    )

    fold_ids = np.full(
        len(dataset),
        -1,
        dtype=np.int64,
    )

    for fold, (_, held_out_indices) in enumerate(
        splitter.split(
            X=np.zeros(len(dataset)),
            y=dataset.labels,
            groups=dataset.groups,
        )
    ):
        fold_ids[held_out_indices] = fold

    test_indices = np.flatnonzero(
        fold_ids == 0
    )
    validation_indices = np.flatnonzero(
        fold_ids == 1
    )
    training_indices = np.flatnonzero(
        fold_ids >= 2
    )

    train_groups = set(
        dataset.groups[training_indices]
    )
    validation_groups = set(
        dataset.groups[validation_indices]
    )
    test_groups = set(
        dataset.groups[test_indices]
    )

    assert train_groups.isdisjoint(
        validation_groups
    )
    assert train_groups.isdisjoint(
        test_groups
    )
    assert validation_groups.isdisjoint(
        test_groups
    )

    split_names = np.full(
        len(dataset),
        "",
        dtype=object,
    )
    split_names[training_indices] = "train"
    split_names[validation_indices] = "validation"
    split_names[test_indices] = "test"

    manifest = pd.DataFrame(
        {
            "filename": dataset.file_names,
            "clip_id": dataset.clip_ids,
            "label": dataset.labels,
            "source_url": dataset.groups,
            "quality": dataset.qualities,
            "split": split_names,
        }
    )

    manifest.to_csv(
        OUTPUT_DIR / "split_manifest.csv",
        index=False,
    )

    for split_name, indices in [
        ("train", training_indices),
        ("validation", validation_indices),
    ]
```

```

        ("test", test_indices),
    ]:
        counts = np.bincount(
            dataset.labels[indices],
            minlength=3,
        )
        print(
            split_name,
            "samples:",
            len(indices),
            "class counts:",
            counts.tolist(),
        )

    return (
        training_indices,
        validation_indices,
        test_indices,
    )

def make_loaders(
    dataset,
    training_indices,
    validation_indices,
    test_indices,
    seed,
):
    generator = torch.Generator()
    generator.manual_seed(seed)

    training_loader = DataLoader(
        Subset(dataset, training_indices),
        batch_size=BATCH_SIZE,
        shuffle=True,
        generator=generator,
        num_workers=0,
        drop_last=True,
    )

    validation_loader = DataLoader(
        Subset(dataset, validation_indices),
        batch_size=BATCH_SIZE,
        shuffle=False,
        num_workers=0,
    )

    test_loader = DataLoader(
        Subset(dataset, test_indices),
        batch_size=BATCH_SIZE,
        shuffle=False,
        num_workers=0,
    )

    return (
        training_loader,
        validation_loader,
        test_loader,
    )

def evaluate(
    model,
    loader,
    criterion,
    device,
):
    model.eval()

    total_loss = 0.0
    total_samples = 0
    all_labels = []
    all_predictions = []

    with torch.no_grad():
        for features, labels in loader:
            features = features.to(device)
            labels = labels.to(device)

            logits = model(features)
            loss = criterion(logits, labels)

            batch_size = labels.size(0)
            total_loss += (
                loss.item() * batch_size

```

```
        )
        total_samples += batch_size

        predictions = logits.argmax(dim=1)

        all_labels.extend(
            labels.cpu().numpy().tolist()
        )
        all_predictions.extend(
            predictions.cpu().numpy().tolist()
        )

    accuracy = accuracy_score(
        all_labels,
        all_predictions,
    )
    macro_f1 = f1_score(
        all_labels,
        all_predictions,
        average="macro",
        zero_division=0,
    )
    balanced_accuracy = (
        balanced_accuracy_score(
            all_labels,
            all_predictions,
        )
    )

    report = classification_report(
        all_labels,
        all_predictions,
        labels=[0, 1, 2],
        output_dict=True,
        zero_division=0,
    )

    matrix = confusion_matrix(
        all_labels,
        all_predictions,
        labels=[0, 1, 2],
    )

    return {
        "loss": total_loss / total_samples,
        "accuracy": accuracy,
        "macro_f1": macro_f1,
        "balanced_accuracy": (
            balanced_accuracy
        ),
        "classification_report": report,
        "confusion_matrix": matrix.tolist(),
    }

def train_one_run(
    sequence_length,
    use_attention,
    seed,
    training_indices,
    validation_indices,
    test_indices,
    device,
):
    set_seed(seed)

    dataset = FencingDataset(
        npy_dir=str(NPY_DIR),
        label_csv=str(LABEL_CSV),
        sequence_length=sequence_length,
    )

    (
        training_loader,
        validation_loader,
        test_loader,
    ) = make_loaders(
        dataset,
        training_indices,
        validation_indices,
        test_indices,
        seed,
    )

    if use_attention:
```

```

        model = AttentionModel().to(device)
        model_type = "attention"
    else:
        model = NoAttentionModel().to(device)
        model_type = "no_attention"

    parameter_count = sum(
        parameter.numel()
        for parameter in model.parameters()
    )

    training_labels = (
        dataset.labels[training_indices]
    )
    class_counts = np.bincount(
        training_labels,
        minlength=3,
    )

    class_weights = (
        len(training_labels)
        / (3.0 * class_counts)
    )

    class_weights_tensor = torch.tensor(
        class_weights,
        dtype=torch.float32,
        device=device,
    )

    criterion = nn.CrossEntropyLoss(
        weight=class_weights_tensor
    )

    optimizer = optim.Adam(
        model.parameters(),
        lr=LEARNING_RATE,
        weight_decay=WEIGHT_DECAY,
    )

    best_validation_accuracy = -1.0
    best_epoch = -1
    best_state = None
    history = []

    first_training_accuracy = None
    first_validation_accuracy = None
    last_training_accuracy = None
    last_validation_accuracy = None

    for epoch in range(1, EPOCHS + 1):
        model.train()

        training_loss_sum = 0.0
        training_correct = 0
        training_samples = 0

        for features, labels in training_loader:
            features = features.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()

            logits = model(features)
            loss = criterion(logits, labels)

            loss.backward()
            optimizer.step()

            batch_size = labels.size(0)

            training_loss_sum += (
                loss.item() * batch_size
            )
            training_samples += batch_size

            predictions = logits.argmax(dim=1)
            training_correct += (
                predictions == labels
            ).sum().item()

        training_loss = (
            training_loss_sum
            / training_samples
        )

```

```

training_accuracy = (
    training_correct
    / training_samples
)

validation_metrics = evaluate(
    model,
    validation_loader,
    criterion,
    device,
)

if epoch == 1:
    first_training_accuracy = training_accuracy
    first_validation_accuracy = (
        validation_metrics["accuracy"]
    )

last_training_accuracy = (
    training_accuracy
)
last_validation_accuracy = (
    validation_metrics["accuracy"]
)

history.append(
    {
        "seed": seed,
        "sequence_length": (
            sequence_length
        ),
        "attention": use_attention,
        "epoch": epoch,
        "training_loss": training_loss,
        "training_accuracy": (
            training_accuracy
        ),
        "validation_loss": (
            validation_metrics["loss"]
        ),
        "validation_accuracy": (
            validation_metrics[
                "accuracy"
            ]
        ),
        "validation_macro_f1": (
            validation_metrics[
                "macro_f1"
            ]
        ),
    }
)

if (
    validation_metrics["accuracy"]
    > best_validation_accuracy
):
    best_validation_accuracy = (
        validation_metrics["accuracy"]
    )
    best_epoch = epoch
    best_state = copy.deepcopy(
        model.state_dict()
    )

print(
    f"seed={seed} "
    f"frames={sequence_length} "
    f"attention={use_attention} "
    f"epoch={epoch:02d} "
    f"train={training_accuracy:.4f} "
    f"val={validation_metrics['accuracy']:.4f} "
    f"val_macro_f1="
    f"{validation_metrics['macro_f1']:.4f}"
)

model.load_state_dict(best_state)

# The test set is evaluated only here,
# after checkpoint selection is complete.
test_metrics = evaluate(
    model,
    test_loader,
    criterion,
    device,
)

```

```

)

run_name = (
    f"frames_{sequence_length}_",
    f"{model_type}_seed_{seed}"
)

history_path = (
    OUTPUT_DIR
    / "histories"
    / f"{run_name}.csv"
)
pd.DataFrame(history).to_csv(
    history_path,
    index=False,
)

checkpoint_path = (
    OUTPUT_DIR
    / "checkpoints"
    / f"{run_name}.pt"
)
torch.save(
    {
        "model_state_dict": best_state,
        "sequence_length": sequence_length,
        "attention": use_attention,
        "seed": seed,
        "best_epoch": best_epoch,
        "parameter_count": parameter_count,
        "class_weights": (
            class_weights.tolist()
        ),
    },
    checkpoint_path,
)

details_path = (
    OUTPUT_DIR
    / "reports"
    / f"{run_name}.json"
)
with open(
    details_path,
    "w",
    encoding="utf-8",
) as file:
    json.dump(
        {
            "classification_report": (
                test_metrics[
                    "classification_report"
                ]
            ),
            "confusion_matrix": (
                test_metrics[
                    "confusion_matrix"
                ]
            ),
        },
        file,
        indent=2,
    )

return {
    "seed": seed,
    "sequence_length": sequence_length,
    "attention": use_attention,
    "parameter_count": parameter_count,
    "class_weight_0": class_weights[0],
    "class_weight_1": class_weights[1],
    "class_weight_2": class_weights[2],
    "best_epoch": best_epoch,
    "best_validation_accuracy": (
        best_validation_accuracy
    ),
    "epoch_1_training_accuracy": (
        first_training_accuracy
    ),
    "epoch_1_validation_accuracy": (
        first_validation_accuracy
    ),
    "epoch_20_training_accuracy": (
        last_training_accuracy
    ),
}

```

```

        "epoch_20_validation_accuracy": (
            last_validation_accuracy
        ),
        "test_accuracy": (
            test_metrics["accuracy"]
        ),
        "test_macro_f1": (
            test_metrics["macro_f1"]
        ),
        "test_balanced_accuracy": (
            test_metrics[
                "balanced_accuracy"
            ]
        ),
        "test_class_0_precision": (
            test_metrics[
                "classification_report"
            ][0][0]["precision"]
        ),
        "test_class_0_recall": (
            test_metrics[
                "classification_report"
            ][0][0]["recall"]
        ),
        "test_class_1_precision": (
            test_metrics[
                "classification_report"
            ][1][0]["precision"]
        ),
        "test_class_1_recall": (
            test_metrics[
                "classification_report"
            ][1][0]["recall"]
        ),
        "test_class_2_precision": (
            test_metrics[
                "classification_report"
            ][2][0]["precision"]
        ),
        "test_class_2_recall": (
            test_metrics[
                "classification_report"
            ][2][0]["recall"]
        ),
    ),
}

def main():
    OUTPUT_DIR.mkdir(
        parents=True,
        exist_ok=True,
    )
    (OUTPUT_DIR / "histories").mkdir(
        exist_ok=True
    )
    (OUTPUT_DIR / "checkpoints").mkdir(
        exist_ok=True
    )
    (OUTPUT_DIR / "reports").mkdir(
        exist_ok=True
    )

    device = torch.device(
        "cuda"
        if torch.cuda.is_available()
        else "cpu"
    )
    print("Device:", device)

    reference_dataset = FencingDataset(
        npy_dir=str(NPY_DIR),
        label_csv=str(LABEL_CSV),
        sequence_length=75,
    )

    actual_counts = np.bincount(
        reference_dataset.labels,
        minlength=3,
    )
    expected_counts = np.asarray(
        [4337, 4501, 532]
    )

    if not np.array_equal(
        actual_counts,

```

```

        expected_counts,
    ):
        raise RuntimeError(
            f"Label counts are wrong: "
            f"{actual_counts.tolist()}; "
            f"expected "
            f"{expected_counts.tolist()}"
        )

    print(
        "Verified label counts:",
        actual_counts.tolist(),
    )

    (
        training_indices,
        validation_indices,
        test_indices,
    ) = create_split(reference_dataset)

    all_results = []

    for seed in SEEDS:
        for sequence_length in SEQUENCE_LENGTHS:
            for use_attention in (
                ATTENTION_OPTIONS
            ):
                result = train_one_run(
                    sequence_length=(
                        sequence_length
                    ),
                    use_attention=(
                        use_attention
                    ),
                    seed=seed,
                    training_indices=(
                        training_indices
                    ),
                    validation_indices=(
                        validation_indices
                    ),
                    test_indices=test_indices,
                    device=device,
                )

                all_results.append(result)

                pd.DataFrame(
                    all_results
                ).to_csv(
                    OUTPUT_DIR
                    / "run_results.csv",
                    index=False,
                )

    results_df = pd.DataFrame(all_results)

    metric_columns = [
        "best_validation_accuracy",
        "epoch_1_training_accuracy",
        "epoch_1_validation_accuracy",
        "epoch_20_training_accuracy",
        "epoch_20_validation_accuracy",
        "test_accuracy",
        "test_macro_f1",
        "test_balanced_accuracy",
        "test_class_0_precision",
        "test_class_0_recall",
        "test_class_1_precision",
        "test_class_1_recall",
        "test_class_2_precision",
        "test_class_2_recall",
    ]

    aggregate = (
        results_df
        .groupby(
            [
                "sequence_length",
                "attention",
            ]
        )[metric_columns]
        .agg(["mean", "std"])
        .reset_index()
    )

```

```
aggregate.columns = [
    "-".join(
        str(part)
        for part in column
        if str(part)
    )
    if isinstance(column, tuple)
    else column
    for column in aggregate.columns
]

aggregate.to_csv(
    OUTPUT_DIR
    / "aggregate_results.csv",
    index=False,
)

print("\nCompleted all experiments.")
print(aggregate.to_string(index=False))

if __name__ == "__main__":
    main()
```

Appendix A2. data_preprocessv1.py

```

import os
import json
import numpy as np
from tqdm import tqdm

class FencingCategorizedProcessor:
    def __init__(self, input_dir, output_dir):
        self.input_dir = input_dir
        self.output_dir = output_dir
        if not os.path.exists(output_dir):
            os.makedirs(output_dir)

        self.stats = {"perfect": 0, "fixed": 0, "failed": 0}

    def process_all_files(self):
        files = sorted(
            f
            for f in os.listdir(self.input_dir)
            if f.endswith(".json")
        )
        print(f"Start processing {len(files)} document...")

        for filename in tqdm(files, desc="Process"):
            file_path = os.path.join(self.input_dir, filename)
            try:
                with open(file_path, 'r', encoding='utf-8') as f:
                    clip_data = json.load(f)

                is_perfect = self.check_if_perfect(clip_data)

                processed_feature = self.normalize_clip(clip_data)

                if processed_feature is not None:
                    if is_perfect:
                        self.stats["perfect"] += 1
                        prefix = "perfect_"
                    else:
                        self.stats["fixed"] += 1
                        prefix = "fixed_"

                    save_name = prefix + filename.replace(".json", ".npz")
                    np.save(os.path.join(self.output_dir, save_name), processed_feature)

            except Exception as e:
                self.stats["failed"] += 1
                print(f"\n❌ {filename} error: {e}")

        print(f"\n" + "=" * 30)
        print(f"📊 Report:")
        print(f"🌟 Perfect (Perfect): {self.stats['perfect']}")
        print(f"🔧 Fixed (Fixed): {self.stats['fixed']}")
        print(f"😞 Failed (Failed): {self.stats['failed']}")
        print(f"=" * 30)

    def check_if_perfect(self, clip_data):
        if len(clip_data) != 75: return False
        for frame in clip_data:
            if len(frame) != 2: return False
            for fencer in frame:
                if len(fencer) != 25: return False
                # if np.array(fencer)[:2].mean() < 0.3: return False
        return True

    def normalize_clip(self, clip_data):
        if len(clip_data) == 0:
            return None

        clip_frames = []
        valid_frame_count = len(clip_data)

        # Store the last valid centre and scale
        # independently for both fencers.
        previous_centres = [None, None]
        previous_scales = [None, None]

        for frame_index in range(75):
            actual_index = min(
                frame_index,
                valid_frame_count - 1,
            )
            frame = clip_data[actual_index]

```

```
normalized_fencers = []
for fencer_index in range(2):
    valid_structure = (
        fencer_index < len(frame)
        and isinstance(
            frame[fencer_index],
            list,
        )
        and len(frame[fencer_index]) == 25
    )

    if not valid_structure:
        normalized_fencers.append(
            np.zeros(
                (25, 2),
                dtype=np.float32,
            )
        )
        continue

    joints = np.asarray(
        frame[fencer_index],
        dtype=np.float32,
    )

    coords = joints[:, :2]

    if joints.shape[1] >= 3:
        confidence = joints[:, 2]
    else:
        confidence = np.ones(
            25,
            dtype=np.float32,
        )

    # A joint is valid only if it has
    # non-zero coordinates and confidence.
    valid_joints = (
        (confidence > 0.0)
        & np.any(coords != 0, axis=1)
    )

    centre = None
    scale = None

    # Prefer the hip as the centre.
    if valid_joints[8]:
        centre = coords[8].copy()

    # Fall back to the shoulder midpoint.
    elif (
        valid_joints[2]
        and valid_joints[5]
    ):
        centre = (
            coords[2] + coords[5]
        ) / 2.0

    # Otherwise reuse the previous centre.
    elif (
        previous_centres[
            fencer_index
        ] is not None
    ):
        centre = previous_centres[
            fencer_index
        ].copy()

    # Prefer neck-to-hip scale.
    if (
        valid_joints[1]
        and valid_joints[8]
    ):
        candidate_scale = np.linalg.norm(
            coords[1] - coords[8]
        )

        if candidate_scale >= 1.0:
            scale = float(
                candidate_scale
            )

    # Fall back to shoulder width.
```

```

    if (
        scale is None
        and valid_joints[2]
        and valid_joints[5]
    ):
        candidate_scale = np.linalg.norm(
            coords[2] - coords[5]
        )

        if candidate_scale >= 1.0:
            scale = float(
                candidate_scale
            )

    # Otherwise reuse the previous scale.
    if (
        scale is None
        and previous_scales[
            fencer_index
        ] is not None
    ):
        scale = previous_scales[
            fencer_index
        ]

    # If neither current nor previous
    # geometry is valid, mask the fencer.
    if centre is None or scale is None:
        normalized_fencers.append(
            np.zeros(
                (25, 2),
                dtype=np.float32,
            )
        )
        continue

    normalized_points = np.zeros(
        (25, 2),
        dtype=np.float32,
    )

    # Only transform detected joints.
    # Missing [0, 0] joints remain [0, 0].
    normalized_points[valid_joints] = (
        coords[valid_joints] - centre
    ) / scale

    previous_centres[
        fencer_index
    ] = centre
    previous_scales[
        fencer_index
    ] = scale

    normalized_fencers.append(
        normalized_points
    )

    clip_frames.append(
        normalized_fencers
    )

    return np.asarray(
        clip_frames,
        dtype=np.float32,
    )
)

processor = FencingCategorizedProcessor(
    r"F:\PycharmProjects\Fencing-Testing"
    r"\dataset\FencersKeyPoints",
    r"F:\PycharmProjects\Fencing-Testing"
    r"\fencing_features_npy_experiment_v2",
)
processor.process_all_files()

```

Appendix A3. dataset.py

```

import os
import re
from pathlib import Path

import numpy as np
import pandas as pd
import torch
from torch.utils.data import Dataset

LABEL_MAP = {
    "1-0-0": 0, # left fencer scores
    "0-0-1": 1, # right fencer scores
    "0-1-0": 2, # no valid score
}

def extract_clip_id(filename):
    stem = Path(filename).stem
    return re.sub(r"^(perfect_|fixed_)", "", stem)

class FencingDataset(Dataset):
    def __init__(
        self,
        npy_dir,
        label_csv,
        sequence_length=75,
        file_list=None,
        transform=None,
    ):
        if sequence_length not in (25, 50, 75):
            raise ValueError(
                "sequence_length must be 25, 50, or 75"
            )

        self.npy_dir = npy_dir
        self.sequence_length = sequence_length
        self.transform = transform

        df = pd.read_csv(label_csv)
        df["ClipName"] = df["ClipName"].astype(str)

        if df["ClipName"].duplicated().any():
            duplicates = df.loc[
                df["ClipName"].duplicated(),
                "ClipName",
            ].tolist()
            raise ValueError(
                f"Duplicate ClipName values: {duplicates[:5]}"
            )

        metadata = df.set_index("ClipName")

        if file_list is None:
            file_names = sorted(
                f
                for f in os.listdir(npy_dir)
                if f.endswith(".npy")
            )
        else:
            file_names = sorted(file_list)

        records = []

        for filename in file_names:
            clip_id = extract_clip_id(filename)

            if clip_id not in metadata.index:
                raise KeyError(
                    f"No CSV row for {filename}; "
                    f"parsed ClipName={clip_id}"
                )

            row = metadata.loc[clip_id]
            raw_label = str(row["Hit_label"]).strip()

            if raw_label not in LABEL_MAP:
                raise ValueError(
                    f"Unknown Hit_label {raw_label!r} "
                    f"for {filename}"
                )

```

```

        records.append(
            {
                "filename": filename,
                "clip_id": clip_id,
                "label": LABEL_MAP[row_label],
                "group": str(row["URL"]),
                "quality": (
                    "perfect"
                    if filename.startswith("perfect_")
                    else "fixed"
                ),
            }
        )
    )

    self.file_names = [
        record["filename"] for record in records
    ]
    self.clip_ids = [
        record["clip_id"] for record in records
    ]
    self.labels = np.asarray(
        [record["label"] for record in records],
        dtype=np.int64,
    )
    self.groups = np.asarray(
        [record["group"] for record in records],
        dtype=object,
    )
    self.qualities = [
        record["quality"] for record in records
    ]

def __len__(self):
    return len(self.file_names)

def __getitem__(self, idx):
    filename = self.file_names[idx]
    file_path = os.path.join(self.npy_dir, filename)

    data = np.load(file_path)

    expected_shape = (75, 2, 25, 2)
    if data.shape != expected_shape:
        raise ValueError(
            f"{filename}: expected {expected_shape}, "
            f"got {data.shape}"
        )

    # Always derive 25/50/75-frame inputs from
    # the same 75-frame base file.
    data = data[-self.sequence_length:]

    x = torch.from_numpy(data).float()
    y = torch.tensor(
        self.labels[idx],
        dtype=torch.long,
    )

    if self.transform is not None:
        x = self.transform(x)

    return x, y

```

Appendix A4. modelv1.py (Attention Model)

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class FencingLSTM(nn.Module):
    def __init__(self, input_size=100, hidden_size=256, num_layers=2, num_classes=3):
        super(FencingLSTM, self).__init__()

        self.bn_input = nn.BatchNorm1d(input_size)

        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
                             batch_first=True, bidirectional=True, dropout=0.4)

        self.attention = nn.Sequential(
            nn.Linear(hidden_size * 2, 128),
            nn.Tanh(),
            nn.Linear(128, 1)
        )

        self.fc = nn.Sequential(
            nn.Linear(hidden_size * 2, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(128, num_classes)
        )

    def forward(self, x):
        batch_size, seq_len, _, _ = x.size()

        x = x.view(batch_size, seq_len, -1).transpose(1, 2)
        x = self.bn_input(x).transpose(1, 2)

        lstm_out, _ = self.lstm(x)

        attn_weights = self.attention(lstm_out) # (batch, 75, 1)
        attn_weights = F.softmax(attn_weights, dim=1)

        context_vector = torch.sum(attn_weights * lstm_out, dim=1)

        logits = self.fc(context_vector)
        return logits
```

Appendix A5. modelv12.py (No-Attention Model)

```
import torch
import torch.nn as nn

class FencingLSTM(nn.Module):
    def __init__(self, input_size=100, hidden_size=256, num_layers=2, num_classes=3):
        super(FencingLSTM, self).__init__()

        self.bn_input = nn.BatchNorm1d(input_size)

        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
                             batch_first=True, bidirectional=True, dropout=0.4)

        self.fc = nn.Sequential(
            nn.Linear(hidden_size * 2, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(128, num_classes)
        )

    def forward(self, x):
        batch_size, seq_len, _, _ = x.size()

        x = x.view(batch_size, seq_len, -1).transpose(1, 2)
        x = self.bn_input(x).transpose(1, 2)

        _, (hidden_state, _) = self.lstm(x)

        forward_final = hidden_state[-2]
        backward_final = hidden_state[-1]

        context_vector = torch.cat(
            [forward_final, backward_final],
            dim=1,
        )

        logits = self.fc(context_vector)
        return logits
```

Appendix A6. Complete 30-Run Summary

The complete run-level validation and archived test results are reported in **Table A1**.

Table A1. Complete run-level validation and archived test results.

Seed	Frames	Attention	Best epoch	Best val. (%)	Final val. (%)	Test acc. (%)
42	25	No	16	56.92	56.55	56.29
42	25	Yes	19	58.71	57.37	57.63
42	50	No	20	58.71	58.71	59.27
42	50	Yes	19	56.25	55.36	56.29
42	75	No	20	59.82	59.82	60.83
42	75	Yes	18	56.62	56.47	56.52
52	25	No	19	58.04	58.04	57.19
52	25	Yes	19	58.48	56.18	58.97
52	50	No	19	56.40	53.05	56.96
52	50	Yes	14	56.62	51.79	55.77
52	75	No	19	56.55	51.64	57.56
52	75	Yes	16	56.62	55.58	53.76
62	25	No	20	58.56	58.56	59.27
62	25	Yes	18	59.52	57.51	57.19
62	50	No	19	60.19	58.56	60.24
62	50	Yes	19	57.07	56.25	56.81
62	75	No	20	57.89	57.89	58.23
62	75	Yes	18	54.02	53.94	52.94
72	25	No	16	60.34	57.37	59.87
72	25	Yes	20	59.38	59.38	55.40
72	50	No	19	59.30	57.74	59.94
72	50	Yes	20	57.29	57.29	54.65
72	75	No	18	59.30	57.29	59.57
72	75	Yes	20	56.25	56.25	58.53
82	25	No	18	59.90	58.85	57.78
82	25	Yes	18	58.56	57.29	57.56
82	50	No	18	59.15	57.44	60.01
82	50	Yes	20	57.66	57.66	57.86
82	75	No	17	58.56	57.14	59.57
82	75	Yes	11	52.46	51.79	50.34