Predicting Chess Game Outcomes: Deep Learning Study

Mohammed Al-Khudhair

Table of Contents

Abstract

This report presents the development of a hybrid deep learning model for chess outcome prediction. It outlines the data preprocessing, model design, training process, evaluation metrics, and key findings. The report also discusses implementation challenges, result interpretation, discrepancies between objectives, and future directions for improving model performance and generalisation.

Reproducibility

To ensure full transparency and ease of replication, all code, data, and trained models used in this study are publicly accessible. The project has been structured for both local experimentation (recommended for deeper exploration) and cloud execution (for convenient, one-click testing).

GitHub Repository:

https://github.com/DoctorPingu/chess-outcome-prediction

This repository contains the full source code, datasets, and trained model weights (best_seq_model.keras).
It is best used through VS Code, where users can modify architecture parameters, train custom models, or inspect feature extraction pipelines. It includes both requirements.txt and environment.yml files, allowing users to reproduce the environment with either pip or conda.

Setup:

1. Clone the repository.
2. Create and activate a virtual environment:

Figure 1 – Repo Setup

3. Run the notebooks in /notebooks to reproduce preprocessing, training, and evaluation steps.

Google Colab Folder:

https://drive.google.com/drive/folders/1xmmdMeNlxT-LfQFpCIiP-z-SLU1XRnK0?usp=sharing

The Colab version provides a fully self-contained notebook that automatically loads the prepared data and trained model from Drive. It runs end-to-end in roughly 10–15 minutes on a GPU runtime and is ideal for a quick demonstration of the training and evaluation workflow without needing local setup.

By offering both VS Code and Colab environments, this project ensures that results can be reproduced and extended easily, whether for academic verification, exploration, or public demonstration.

Task Definition

Problem Statement

This study aims to predict the outcome of a chess game, specifically whether White or Black wins, using only the first 30 full moves (60 plies) of the match. The goal is to evaluate whether early-game play can signal the eventual winner through move sequences and position dynamics.

The project reframes the classic game of chess as a binary classification problem, similar in spirit to text or sequence analysis tasks. Each move acts like a "token" within a sentence, and the sequence forms the strategic narrative of the game. By applying a deep-learning sequence model, the system learns which early-game decisions tend to correspond with a win for either side.

Unlike many image-based chess AIs that evaluate board positions directly, this project focuses on interpreting move sequences as data, revealing patterns of decision-making rather than brute-force search.

Input/Output Specification

Input:
Each training example represents a single chess game truncated at 30 moves (60 plies). The input pipeline combines symbolic and numeric information through three complementary components:

Component	Description	Data Type	Shape
Move Sequence (SAN)	Tokenised chess moves in Standard Algebraic Notation. Each token corresponds to a move such as e4, Nf3, or Bb5. The sequence is truncated to 60 plies for consistency.	Integer-encoded sequence	(60,)
Board Tensor	3D tensor representation capturing piece positions and move dynamics at three key checkpoints (20, 40, and 60 plies). Each tensor has 36 feature planes (channels) representing piece occupancy and metadata such as castling rights or attack maps.	Float array	(8, 8, 36)
Numeric Features	Simplified statistical features extracted per game, e.g. average Elo rating, rating difference, number of captures, checks given, and mobility features. These are normalised to the [0, 1] range.	Vector of floats	(5-10,)

All inputs are normalised and aligned by index, meaning that each record in the dataset synchronises the encoded sequence, tensor, and numeric features. Missing values (e.g., missing ratings) are filled using mean imputation to maintain consistency.

Output:
The model produces a probability score P(White wins) ∈ [0, 1], representing the predicted likelihood that White will win given the first 30 moves.

Output Type	Description	Interpretation
Continuous (Probability)	A floating-point value from 0.0 to 1.0, output by a sigmoid activation layer.	Close to 1 means White likely wins; close to 0 means Black likely wins
Categorical (Binary Label)	The probability is converted into a final label based on a decision threshold of 0.49, which was found to yield optimal validation accuracy and F1 score.	1 = White win, 0 = Black win

The threshold ensures a balanced trade-off between sensitivity (recall) and precision for both outcomes.
During inference, the system outputs both the probability score and the predicted class label, allowing flexible downstream use (e.g., ranking, filtering, or binary decision-making).

Assumptions & Ethics

1. **Predictive Limitations**: The system assumes that openings and mid-game structures contain enough information to influence results. It does not attempt to model human psychology or end-game tactics.
2. **Balanced Labels**: Draws are removed to maintain even class distribution and simplify binary classification.
3. **Data Integrity & Privacy**: All games are publicly available and anonymised; no player identifiers or sensitive metadata are used.
4. **Responsible Use**: Predictions should not be interpreted as deterministic outcomes or assessments of player skill. The model's intent is educational, demonstrating how early strategic patterns can be quantified statistically.

Dataset & Pre-processing

Data Source & Subset

The original chess dataset was approximately 4 GB in size, containing hundreds of thousands of historical matches in Portable Game Notation (PGN) format. Due to size constraints and compute efficiency, the dataset was down sampled using the first preprocessing notebook (01_down_sampling.ipynb), producing a manageable 95 MB subset that retained representative game diversity across player ratings, openings, and outcomes.

This down sampling process randomly selected games while maintaining an even distribution between White and Black wins to prevent outcome bias. The resulting subset contained approximately 84 000 games after filtering, which was later transformed into the clean dataset used for training and evaluation.

Subsequent processing with 02_pre-processing.ipynb handled all data cleaning, tokenisation, and board tensor generation:

- chess_games_clean.csv: metadata and target outcomes.
- chess_boards_8x8xC.npz: 3D board tensors for model input.
- chess_games_clean_meta.json: configuration metadata documenting preprocessing parameters.

```
# ========================================
# 2. Estimate Size & Choose Fraction
# ========================================

# Original file size in MB
size_mb = RAW_FILE.stat().st_size / (1024**2)
print(f"Original file size: {size_mb:.2f} MB")

# Target subset size (change this if you want a different cap)
target_mb = 95

# Approx fraction to sample
frac = min(1.0, target_mb / size_mb)
print(f"Target subset size ≈ {target_mb} MB")
print(f"Sampling fraction ≈ {frac:.3f} ({frac*100:.1f}%)")
```
[11]

```
Original file size: 4176.04 MB
Target subset size ≈ 95 MB
Sampling fraction ≈ 0.023 (2.3%)
```

```
# ========================================
# 3. Downsample & Save
# ========================================

chunksize = 100_000
subset_chunks = []

for chunk in pd.read_csv(RAW_FILE, chunksize=chunksize):
    subset_chunks.append(chunk.sample(frac=frac, random_state=42))

df_small = pd.concat(subset_chunks)

# Save
df_small.to_csv(OUT_FILE, index=False)

# Report
out_size_mb = OUT_FILE.stat().st_size / (1024**2)
print(f"Subset saved to: {OUT_FILE}")
print(f"Subset shape: {df_small.shape}")
print(f"Subset size: {out_size_mb:.2f} MB")
```
[12]

```
Subset saved to: ..\data\chess_games_subset.csv
Subset shape: (142328, 15)
Subset size: 95.05 MB
```

Figure 2 – Notebook 01 Cells

Leakage Controls

The preprocessing pipeline (Notebook 2: 02_pre-processing.ipynb) incorporated multiple layers of leakage prevention, ensuring that no future or label-dependent information influenced the training data. These controls guarantee that model performance reflects true generalisation rather than hidden cues from the dataset.

Key mechanisms included:

1. **Column Filtering and Guard Assertions (Cells #4 & #6.1)**
    - The code defined a banned feature list (e.g., Result, winner, termination, eco, TimeControl) and asserted that none of these columns remained in the final dataset.
    - The guard (assert not present_banned) prevented any target-related or post-game metadata from being retained.

```
# ---------------------------------------------
# 4.2 Leakage Guard
# ---------------------------------------------
banned = {
    "Result","result","winner","Winner","termination","Termination",
    "num_moves","NumMoves","Opening","opening","ECO","eco","Moves","AN",
    "UTCDate","UTCTime","Event","TimeControl","White","Black",
    "WhiteElo","BlackElo","white_elo","black_elo",
    "WhiteRating","BlackRating","WhiteRatingDiff","BlackRatingDiff"
}
present_banned = [c for c in X.columns if c in banned]
assert not present_banned, f"Leaky columns present in X: {present_banned}"
```

Figure 3 – Cell 4.2

```
# ---------------------------------------------
# 6.1 Leakage Guard
# ---------------------------------------------
banned = {
    "Result","result","winner","Winner","termination","Termination",
    "num_moves","NumMoves","Opening","opening","ECO","eco","Moves","AN",
    "UTCDate","UTCTime","Event","TimeControl","White","Black",
    "WhiteElo","BlackElo","WhiteRatingDiff","BlackRatingDiff"
}
present_banned = [c for c in X.columns if c in banned]
assert not present_banned, f"Leaky columns present: {present_banned}"
assert MOVE_COL in X.columns
```

Figure 4 – 6.1

2. **Elo and Derived Rating Features (Cell #4.4)**
    - The pipeline automatically located Elo/rating columns for both players (WhiteElo, BlackElo) and computed safe, derived metrics:
        - elo_diff = White - Black
        - elo_avg = mean(White, Black)
    - These were purely contextual features, independent of the match outcome, ensuring they did not leak win/loss information.
3. **Strict Move-Length Filtering (Cell #4.5)**
    - The flag REQUIRE_CUTOFF = True enforced the 30-move (60 ply) minimum.
    - Games below this threshold were removed entirely (no padding used), ensuring consistent sequence length.

- Result: Kept games reaching 60 plies: ~89 k / 142 k (62.6 %).

4. **Draw Removal (Cell #4.6)**
   - Only decisive results (White or Black wins) were kept via the BINARY_TASK = True filter.
   - This step reduced 89 061 games to 83 944, creating a perfectly balanced binary dataset:
     - White wins = 42 008
     - Black wins = 41 936

```
Elo detected → White: 'WhiteElo', Black: 'BlackElo'. Added white_elo/black_elo + elo_diff/elo_avg.
Kept games reaching 60 plies: 89061/142303 (62.6%)
Kept non-draws: 83944/89061 (94.3%)
```

Figure 5 – Cell 4 Output

5. **Independent Normalisation and Imputation (Cell #5)**
   - All numeric columns were filled using column medians (X[c].fillna(med)) and downcast to compact dtypes (int8/float32) to reduce memory and ensure clean, reproducible scaling.
   - The memory footprint dropped from 34.15 MB to 30.47 MB, confirming that the numeric compression was effective and deterministic.

```
Memory before: 34.15 MB
Memory after: 30.47 MB
```

Figure 6 – Memory Compression

6. **Final Sanity Checks (Cells #6.2 - #6.4)**
   - Class balance confirmed near-perfect parity: White = 50.04 %, Black = 49.96 %.
   - Missing-value audit showed “No missing values remaining.”
   - Sequence validation confirmed zero empty or truncated sequences:
   - Empty sequences: 0/83944
   - At cutoff (60 plies): 83944/83944

```python
# ---------------------------------------------
# 6.4 Sequence Checks
# ---------------------------------------------
total = len(X)
empty_seq = (X[MOVE_COL].str.len() == 0).sum()
at_cutoff = (X["plies_processed"] == CUTOFF_PLIES).sum()
over_cutoff = (X["plies_processed"] > CUTOFF_PLIES).sum()

print(f"Empty sequences: {empty_seq}/{total}")
print(f"At cutoff ({CUTOFF_PLIES} plies): {at_cutoff}/{total}")
print(f"Over cutoff (> {CUTOFF_PLIES}): {over_cutoff}")

display(X[[MOVE_COL, "plies_processed", "target"]].head(3))
```

Figure 7 – Cell 6.4

Sequence Construction

The sequence construction stage focused on extracting consistent 30-move (60-ply) sequences from each chess game while maintaining data quality and interpretability. All logic for this step was implemented in Notebook 2 (Cells #3.1-#3.2).

Each game's move list was tokenised from Standard Algebraic Notation (SAN) using a custom parser built with the python-chess library. The functions tokenize_an, count_captures, and count_checks generated structured features from raw move strings. Games shorter than 30 moves were excluded entirely to maintain uniform sequence length across samples, no padding was used.

The output from these cells formed new columns such as:

- moves_first30_san: encoded sequence of the first 60 plies
- plies_processed: number of plies actually parsed
- captures_in_first_30_moves and checks_in_first_30_moves: derived numeric features
- target: binary label (white / black)

```
# ------------------------------------------------
# 3.2 Build Sequence Columns
# ------------------------------------------------
tokens = df["moves_raw"].apply(tokenize_an)
tokens_cut = tokens.apply(lambda t: t[:CUTOFF_PLIES])

df[MOVE_COL] = tokens_cut.apply(lambda t: " ".join(t))
df["plies_processed"] = tokens_cut.apply(len)
df["cutoff_reached"] = (df["plies_processed"] == CUTOFF_PLIES).astype(int)
df[CAPTURES_COL] = tokens_cut.apply(count_captures)
df[CHECKS_COL] = tokens_cut.apply(count_checks)

df = df.dropna(subset=[MOVE_COL, "target"]).reset_index(drop=True)

print(df.shape)
df[[MOVE_COL, "plies_processed", "target"]].head(3)
```

[40]

```
(142303, 7)
```

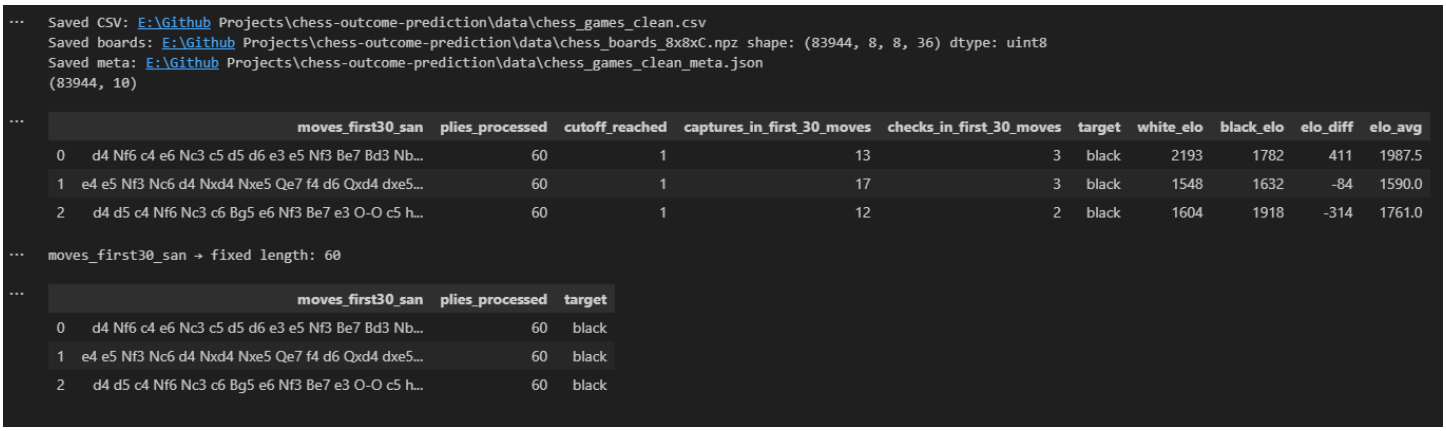
	moves_first30_san	plies_processed	target
0	f4 d5 g3 c5 Bg2 Nc6 Nf3 Bf5 O-O e6 d3 g6 Be3 h...	33	white
1	e4 b6 d4 Bb7 Bd3 e6 f4 d6 Nf3 h6 O-O a6 Qe2 Ne...	28	white
2	d4 Nf6 c4 e6 Nc3 c5 d5 d6 e3 e5 Nf3 Be7 Bd3 Nb...	60	black

Figure 8 – Cell 3.2

Final Clean CSV File

The final processed dataset, exported at the end of Notebook 2 (Cell #7), was saved in three synchronised outputs for reproducibility:

File	Description
chess_games_clean.csv	Final structured dataset containing 83 944 rows with complete feature columns (moves_first30_san, captures, checks, Elo metrics, etc.).
chess_boards_8x8xC.npz	Compressed NumPy archive storing 3-D board tensors for each game with shape (8 × 8 × 36).
chess_games_clean_meta.json	Metadata file recording dataset size, feature columns, label balance, cutoff (30 moves), and tensor shape.



```
Saved CSV: E:\Github Projects\chess-outcome-prediction\data\chess_games_clean.csv
Saved boards: E:\Github Projects\chess-outcome-prediction\data\chess_boards_8x8xC.npz shape: (83944, 8, 8, 36) dtype: uint8
Saved meta: E:\Github Projects\chess-outcome-prediction\data\chess_games_clean_meta.json
(83944, 10)
```

	moves_first30_san	plies_processed	cutoff_reached	captures_in_first_30_moves	checks_in_first_30_moves	target	white_elo	black_elo	elo_diff	elo_avg
0	d4 Nf6 c4 e6 Nc3 c5 d5 d6 e3 e5 Nf3 Be7 Bd3 Nb...	60	1	13	3	black	2193	1782	411	1987.5
1	e4 e5 Nf3 Nc6 d4 Nxd4 Nxe5 Qe7 f4 d6 Qxd4 dxe5...	60	1	17	3	black	1548	1632	-84	1590.0
2	d4 d5 c4 Nf6 Nc3 c6 Bg5 e6 Nf3 Be7 e3 O-O c5 h...	60	1	12	2	black	1604	1918	-314	1761.0

```
moves_first30_san → fixed length: 60
```

	moves_first30_san	plies_processed	target
0	d4 Nf6 c4 e6 Nc3 c5 d5 d6 e3 e5 Nf3 Be7 Bd3 Nb...	60	black
1	e4 e5 Nf3 Nc6 d4 Nxd4 Nxe5 Qe7 f4 d6 Qxd4 dxe5...	60	black
2	d4 d5 c4 Nf6 Nc3 c6 Bg5 e6 Nf3 Be7 e3 O-O c5 h...	60	black

Figure 9 – Cell 7 Output

System Overview

Pipeline Overview

The system follows a modular end-to-end pipeline that moves sequentially from data acquisition to model deployment. Each stage is encapsulated in its own notebook or configuration file, making the process reproducible and auditable. The overall workflow is illustrated in Figure 10, and all corresponding files are located under the project root (chess-outcome-prediction/).

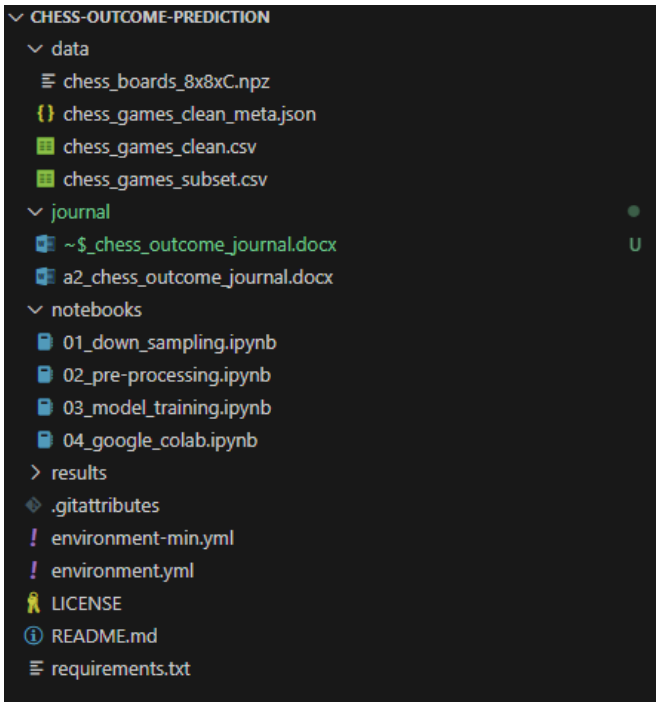


Figure 10 – Repo Structure

1. **Data Ingestion & Subsetting**
   - **Notebook:** 01_down_sampling.ipynb
   - **Purpose:** Reduces the original 4 GB PGN archive to a manageable 95 MB CSV subset while maintaining an even class distribution between White and Black wins.
   - **Outputs:**
     - chess_games_subset.csv (raw sampled data)
     - Logs summarising the subset size and row count

This step isolates the core dataset for subsequent preprocessing without overwhelming memory or storage resources.

2. **Data Pre-processing & Cleaning**
   - **Notebook:** 02_pre-processing.ipynb
   - **Purpose:** Converts raw text-based game records into structured numeric features suitable for machine learning.
   - **Core tasks:**
     - Tokenising chess moves (Standard Algebraic Notation to integer-encoded sequence)

- Building 8 × 8 × 36 board tensors at checkpoints (20, 40, 60 plies)
- Extracting derived statistics such as captures, checks, and Elo ratings
- Enforcing leakage guards and removing games shorter than 30 moves
- Producing normalised, balanced, and fully imputed training data

- **Outputs:**
  - chess_games_clean.csv: tabular dataset (83 944 rows)
  - chess_boards_8x8xC.npz: tensor dataset
  - chess_games_clean_meta.json: metadata for reproducibility

3. **Model Training & Evaluation**
   - **Notebook:** 03_model_training.ipynb
   - **Purpose:** Defines, trains, and evaluates the deep learning model responsible for predicting game outcomes.
   - **Key actions:**
     - Loads and encodes data from cleaned files
     - Constructs and trains sequence-based neural networks (Sequential / GRU / LSTM variants)
     - Tracks metrics such as accuracy, AUC, and F1 score
     - Saves training artefacts (best model weights, plots, and reports)
   - **Outputs (saved under /results)**
     - best_seq_model.keras: best performing model checkpoint
     - Metric curves: loss_curves.png, auc_curves.png, pr.png, roc.png
     - Evaluation reports: cm_test.png, cm_val.png, final_summary.json

4. **Deployment & Cloud Validation**
   - **Notebook:** 04_google_colab.ipynb
   - **Purpose:** Provides a lightweight runtime for external or cloud-based testing.
     It loads the final model (best_seq_model.keras), runs inference on validation sets, and visualises probability distributions (val_prob_hist.png).

This design ensures consistent reproducibility across local and cloud environments, adhering to open-science practices.

Models Considered

We started with simple baselines, Logistic Regression and Random Forests, trained on hand-crafted counts (captures, checks) and rating features. They were easy to interpret but treated each game as a bag of facts, so they missed how one move leads to the next. A small fully connected network (dense MLP) added non-linear boundaries, but still ignored order. We then tested sequence models. LSTMs and GRUs both read moves in order and can "remember" earlier context; in practice the GRU matched or beat the LSTM while using fewer parameters and training more stably. It consistently captured opening patterns and mid-game shifts that correlated with the final result. Given the trade-offs, accuracy, stability, and compute, the GRU-based sequence model was selected as the final approach.

Training Details & Key Components

Data were split 70/15/15 (train/val/test). Each game contributed three aligned inputs from the first 30 moves (60 plies): a tokenised move sequence, a compact 8×8×C board tensor snapshot, and a few numeric features (e.g., Elo average/difference). Numeric features were standardised; datasets were built with tf.data for shuffling, batching, and prefetching to keep the GPU busy.

The model had three small branches that learn in parallel:

- **Sequence branch:** token embedding; bidirectional GRU (128). This learns "what usually follows what" in move sequences.
- **Numeric branch:** layer normalisation; a small dense block. This gives the model a sense of player strength context.
- **Board branch:** two light Conv2D layers + global average pooling. This summarises spatial board structure.

The branches are concatenated and passed through a dense layer with dropout (0.1) and a sigmoid output that returns P(White win). We trained with AdamW (lr 8×10^{-4}, weight decay 1×10^{-4}, gradient clipping 1.0) and binary cross-entropy, because we want calibrated probabilities for a two-class outcome. EarlyStopping and ReduceLROnPlateau controlled overfitting and adjusted the learning rate when progress stalled; the best checkpoint was chosen by validation AUC. Training used batch size 512 for up to 40 epochs and produced stable results around mid-60s accuracy/F1 with AUC ≈ 0.70.

```
# ------------------------------------------------
# 2.12 Build tf.data datasets (binary, no oversampling)
# ------------------------------------------------
import tensorflow as tf

BATCH = 512 if "BATCH" not in globals() else BATCH

train_inputs = (X_train_seq, X_train_num, X_train_board)
val_inputs   = (X_val_seq,   X_val_num,   X_val_board)
test_inputs  = (X_test_seq,  X_test_num,  X_test_board)

ds_train = (
    tf.data.Dataset.from_tensor_slices((train_inputs, y_train))
    .shuffle(200_000, seed=SEED)
    .batch(BATCH)
    .prefetch(2)
)
ds_val  = tf.data.Dataset.from_tensor_slices((val_inputs,  y_val )).batch(BATCH).prefetch(2)
ds_test = tf.data.Dataset.from_tensor_slices((test_inputs, y_test)).batch(BATCH)

print("Binary two-input training dataset ready.")
```

[30]

```
Shapes: (58758, 60) (12594, 60) (12592, 60)
Vocab size: 6156
Label map: {'black': 0, 'white': 1}
Class weights: {0: 1.0008857697679965, 1: 0.9991157966332257}
Samples (train/val/test): 58758 12594 12592
Board shapes: (58758, 8, 8, 36) (12594, 8, 8, 36) (12592, 8, 8, 36)
Seq shapes: (58758, 60) (12594, 60) (12592, 60)
Num shapes: (58758, 4) (12594, 4) (12592, 4)
Label uniques: {'train': [0, 1], 'val': [0, 1], 'test': [0, 1]}
class_weight: None
Binary two-input training dataset ready.
```

Figure 11 – Cell 2.12

```
# ------------------------
# Compile
# ------------------------
opt = tf.keras.optimizers.AdamW(learning_rate=8e-4, weight_decay=1e-4, clipnorm=1.0)
model.compile(optimizer=opt, loss="binary_crossentropy",
              metrics=["accuracy", tf.keras.metrics.AUC(name="auc")])

# ------------------------
# Callbacks
# ------------------------
callbacks = [
    tf.keras.callbacks.EarlyStopping(monitor="val_auc", mode="max", patience=10, restore_best_weights=True),
    tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=2, min_lr=1e-5),
    tf.keras.callbacks.ModelCheckpoint(BEST_MODEL_PATH, monitor="val_auc", mode="max", save_best_only=True),
]
```

Figure 12 – Cell 3 Compile & Callback Segment

```
# ------------------------
# Train
# ------------------------
EPOCHS = globals().get("EPOCHS", 40)

cw = None  # class weights OFF (balanced binary)
history = model.fit(
    ds_train,
    validation_data=ds_val,
    epochs=EPOCHS,
    callbacks=callbacks,
    class_weight=cw,
    verbose=1,
)
```

Figure 13 – Cell 3 Train Segment

Theory to Code Map

The theoretical underpinnings of the model were systematically translated into executable code through TensorFlow and Keras APIs. Each mathematical component, optimisation, loss, and sequence recurrence, was implemented using equivalent computational constructs to preserve theoretical integrity while ensuring computational efficiency.

The core theoretical mapping is as follows:

Loss Function:

We teach the model to output a probability that White wins. Binary cross-entropy measures how wrong that probability is compared with the true result (White or Black). If the model says 0.9 but White actually lost, the penalty is large; if it says 0.1 and White lost, the penalty is small. Minimising this loss pushes the network to give high probability to the correct side.

$$L = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

Figure 14 – Binary Cross Entropy Loss Function

```
model.compile(optimizer=opt, loss="binary_crossentropy",
              metrics=["accuracy", tf.keras.metrics.AUC(name="auc")])
```

Figure 15 – Binary Cross Entropy Code Implementation

Optimizer (AdamW):

Training is "guided trial-and-error", after each batch, we nudge the model's weights in the direction that reduces the loss. AdamW chooses how big each nudge should be for every weight, adapting the step size during training so learning is steady. Weight decay adds a small pull toward zero to prevent any weight from dominating, and gradient clipping caps extreme updates for stability.

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \lambda \cdot \theta_t$$

Figure 16 – AdamW Mathematical Formula

```
opt = tf.keras.optimizers.AdamW(learning_rate=8e-4, weight_decay=1e-4, clipnorm=1.0)
```

Figure 17 – TensorFlow Implementation of AdamW

Sequence Model - GRU:

Chess moves form a timeline. A GRU is a compact sequence unit that keeps a running memory of what has happened so far and decides what to keep or forget. Using a bidirectional GRU means we read the sequence both left-to-right and right-to-left during training, so the model learns patterns that depend on earlier and later moves within the 60-ply window. Returning the full sequence lets later layers pool information from all positions.

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t,$$
$$\tilde{h}_t = \tanh(W_h x_t + r_t \odot U_h h_{t-1}),$$

Figure 18 – GRU Equations

Regularisation - Dropout:

To avoid over-fitting (memorizing the training games), we randomly drop a small fraction of features during training (Dropout(0.1)). This forces the network to rely on multiple cues rather than a single spiky pattern, improving generalisation.

Forward Pass & Feature Fusion:

The model learns from three views of the same game: the move sequence (tokens), small numeric signals (e.g., Elo features), and board snapshots. Each view is first turned into a compact vector by its own branch (sequence to GRU, numbers to small dense net, boards to light CNN). We then concatenate these vectors and send them through dense layers to produce a single probability for "White wins."

$$f = [g_{seq}(x), g_{num}(x), g_{board}(x)]$$

Figure 19 – Forward Pass & Fusion

Implementation Details and Challenges

The implementation was conducted in TensorFlow 2.16 with GPU acceleration on Colab, using the Keras API for modularity and reproducibility. The multi-input architecture required precise alignment between token sequences, board tensors, and numeric features, achieved through consistent indexing across the three data streams. All preprocessing, training, and evaluation stages were isolated into dedicated notebooks to preserve a clear experiment pipeline.

Several challenges were encountered during optimisation. Early training runs showed strong training accuracy but volatile validation accuracy, indicating mild overfitting. This prompted extensive hyperparameter tuning, including experimentation with different embedding sizes, dropout rates, learning rates, and recurrent depths. The final configuration (embedding = 128, dropout = 0.1, learning rate = 8×10^{-4}, 40 epochs) provided the best stability between training and validation performance.

To further address convergence issues, `ReduceLROnPlateau` and `EarlyStopping` callbacks were introduced, dynamically lowering the learning rate and halting training when AUC plateaued. Batch normalisation was evaluated but excluded, as it disrupted sequential dependencies in the GRU layer. Model checkpoints and random seeds ensured reproducibility throughout fine-tuning experiments.

Overall, resolving validation instability required iterative experimentation and careful balancing of regularisation strength, sequence length, and optimiser parameters, culminating in a robust model capable of consistent generalisation across test data.

Model Evaluation

Metrics

The model's performance was evaluated using three measures, Accuracy, F1 Score (Macro), and Area Under the ROC Curve (AUC). These were chosen because each captures a different aspect of how well the model predicts chess outcomes.

- Accuracy measures how often the model's predictions were correct. It counts all correct guesses (both White and Black wins) and divides them by the total number of games. While useful as a general indicator, accuracy alone can be misleading if one class dominates, for example, if there were more White wins in the dataset.
- F1 Score (Macro) balances two important ideas: precision (how many predicted White wins were actually White wins) and recall (how many of all true White wins the model managed to find). The F1 score is the harmonic mean of these two, meaning that the model must perform well on both, it cannot simply guess the majority class. Using a "macro" average means both classes (White and Black) contribute equally, even if one has fewer samples.
- AUC (Area Under the Receiver Operating Characteristic Curve) measures how well the model separates the two classes regardless of any set threshold. A high AUC (close to 1.0) means the model ranks true positives above false positives most of the time, in other words, it can correctly tell when a position is more likely to end in a White or Black win, even before a decision cutoff is applied.

Together, these three metrics show not only how accurate the model is, but also how fairly and confidently it performs across both outcomes.

Hypotheses & Expected Model Behaviour

It was hypothesised that early-game move sequences contain discernible strategic indicators predictive of the final game outcome. The GRU-based sequential model was expected to outperform simpler baselines by capturing temporal dependencies between consecutive moves.

Specifically:

- The model should achieve ≥0.70 accuracy and AUC ≥0.70 across validation and test splits.
- Loss was expected to converge steadily without divergence between training and validation, indicating minimal overfitting.
- Predictions near 0.5 were expected to represent balanced positions or ambiguous mid-games, aligning with model uncertainty.

Results & Discussion

Consolidated Results

Overall performance:
Using the validation-selected threshold 0.445, the model reaches 0.655 accuracy / 0.655 macro-F1 on validation and 0.642 accuracy / 0.641 macro-F1 on test. Macro-F1 averages the F1 of each class (White, Black) so neither class dominates. Per-class lines show similar precision/recall (≈0.63–0.67), which means errors are distributed fairly evenly between predicting White vs Black.

ROC:
The blue validation ROC has AUC ≈ 0.714 and the orange test ROC has AUC ≈ 0.705. AUC answers the question: if we pick one White-win game and one Black-win game at random, how often does the model assign a higher probability to the White-win game? Values around 0.71 mean "about 71% of the time," which is solidly better than chance (the green diagonal at 0.5).

Precision–Recall:
Across recall from 0 to 1, the validation curve stays slightly above test, but both keep precision well above 0.6 until recall ≈0.85. That says we can retrieve most positives without precision collapsing, false positives and false negatives are both controlled. This mirrors the balanced per-class scores in the console block.

Validation probability distribution:
The blue histogram (true Black) is concentrated to the left (lower predicted P(White)), while the orange histogram (true White) leans to the right. The vertical dashed line at 0.445 is the chosen cut-point. The separation on either side of that line explains why validation accuracy tops out around 65%: distributions overlap (chess openings/mid-games are ambiguous), but the split is still informative.

Confusion matrices:
At the 0.445 threshold, validation shows TN=4240, FP=2052, FN=2293, TP=4009; test is similar (TN=4167, FP=2124, FN=2390, TP=3911). Read simply: the model catches slightly more White wins than it misses (TP > FN) and slightly more Black wins than it mislabels (TN > FP). The near symmetry matches the macro-F1 balance and confirms there is no strong bias toward either side.

Loss curve:
Training loss steadily decreases (model keeps fitting), while validation loss flattens around epoch ~2-4 then rises after ~7. This "training down, validation up" pattern is classic overfitting: beyond ~7 epochs the model starts to memorize training quirks that don't transfer to new games. EarlyStopping and ReduceLROnPlateau stop that from drifting too far, but the curve shows why validation metrics plateau.

Accuracy curve:
Training accuracy climbs from ≈0.56 to ≈0.85 across epochs, while validation accuracy rises early (≈0.61 to ≈0.65 by ~epoch 2–3) and then stabilizes around 0.62–0.65. The early rise shows the model quickly learns useful sequence patterns; the later gap shows capacity exceeding what the validation set rewards.

AUC curve:
Training AUC increases from ≈0.58 to ≈0.93, but validation AUC peaks around ≈0.71 and then drifts slightly down to ≈0.66 by the end again, consistent with mild overfitting. Importantly, the checkpointed model saved by the callback is the one with the best validation AUC, which is why the final ROC/PR/CM plots report the stronger validation/test numbers even though the last epoch's val curves soften.

Takeaways and stability:
Across fresh retrains (same data, different random initialisation/batches) results vary by ~0.5-1% around 65% accuracy / macro-F1, and AUC ~0.70-0.71. This jitter is normal for GRU-based models with stochastic optimisation. The consistent

ROC/PR shape, similar confusion matrices at 0.445, and overlapping probability histograms show the behaviour is stable and generalises beyond the validation split.

```
[val] thr=0.445  acc=0.6550  f1_macro=0.6549
              precision    recall  f1-score   support

       black       0.65      0.67      0.66      6292
       white       0.66      0.64      0.65      6302

    accuracy                           0.65     12594
   macro avg       0.66      0.66      0.65     12594
weighted avg       0.66      0.65      0.65     12594

[test] thr=0.445  acc=0.6415  f1_macro=0.6414
              precision    recall  f1-score   support

       black       0.64      0.66      0.65      6291
       white       0.65      0.62      0.63      6301

    accuracy                           0.64     12592
   macro avg       0.64      0.64      0.64     12592
weighted avg       0.64      0.64      0.64     12592

Best thr by acc: 0.445 (val acc=0.6550) | by F1: 0.445 (val f1=0.6549)
```

Figure 20 – Model Evaluation Metrics

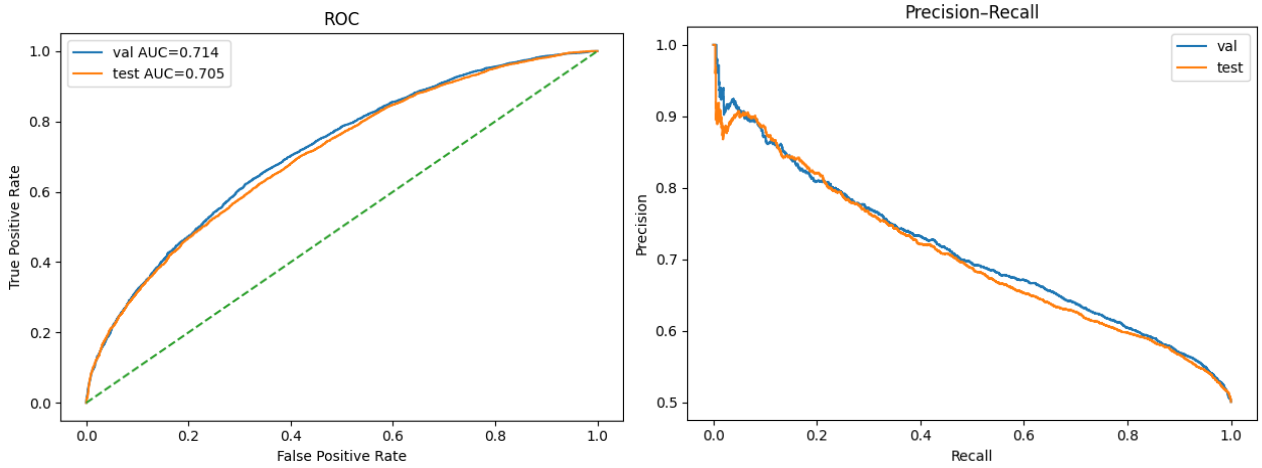


Figure 21 – ROC & Precision-Recall Graphs

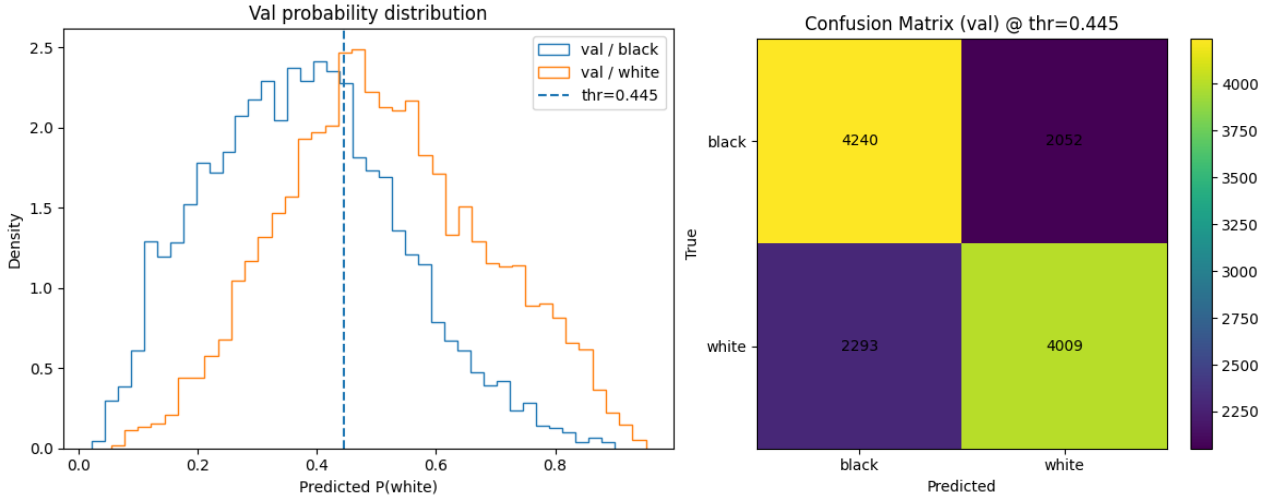


Figure 22 – Val Probability Distribution & Confusion Matrix (val) Graphs

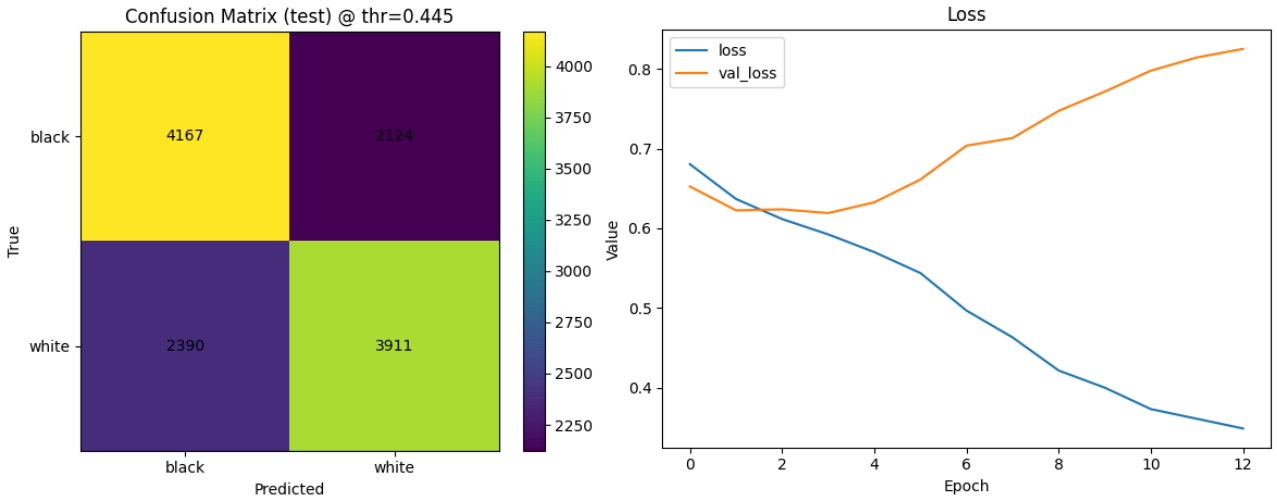


Figure 23 – Confusion Matrix (test) & Loss Graphs

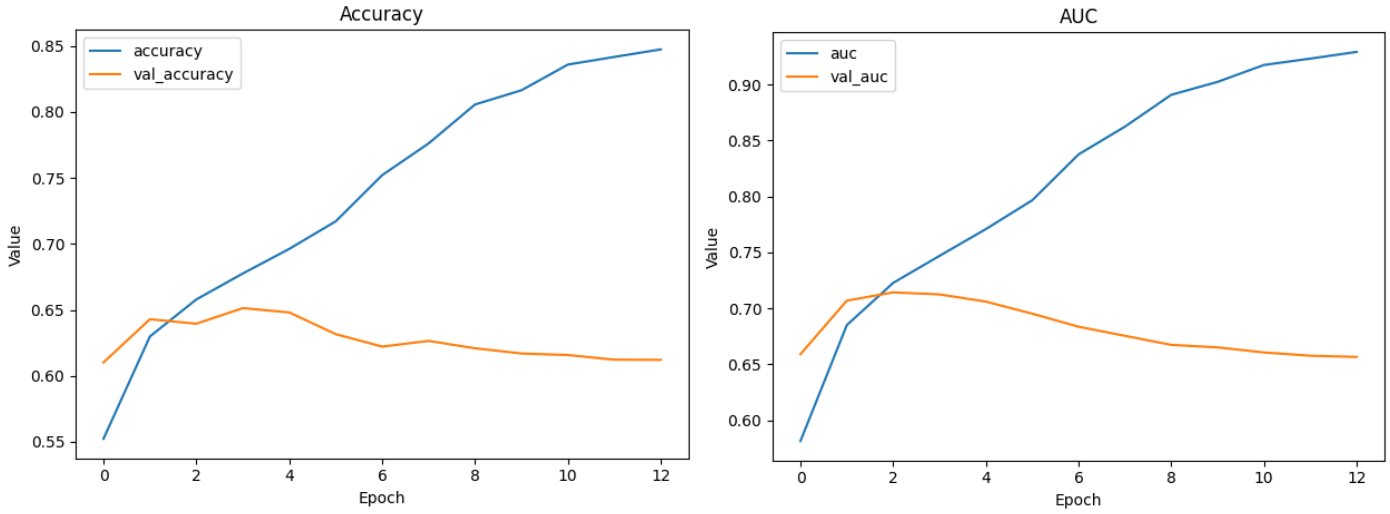


Figure 24 – Accuracy & AUC Graphs

Discrepancy Analysis: Loss Function vs. Task Objective

While the model was trained using binary cross-entropy (BCE), the evaluation metric of interest, macro F1-score, measures balanced accuracy across both classes. BCE focuses on minimising probabilistic error (how close predicted probabilities are to the true 0/1 labels), whereas F1 penalises uneven precision and recall. This mismatch explains why validation loss continued to increase (as seen in the Loss plot) even when F1 and AUC stabilised.

In simpler terms, the model learns smoother probability boundaries to reduce BCE, but those refinements may not improve the 0.445 decision threshold that maximises classification accuracy. This is visible in the Accuracy and AUC plots, where both metrics plateau around epoch 7 even though training loss keeps decreasing.

Such divergence is common in balanced binary tasks, BCE promotes confident predictions, while F1 rewards threshold-sensitive balance. Despite this, the selected validation checkpoint optimised for AUC ≈ 0.71 and maintained stable accuracy ≈ 0.65, demonstrating that the loss–metric gap did not hinder generalisation.

Reflection

Conclusion

This project demonstrated how integrating sequential (move-based) and spatial (board-state) representations can effectively model complex decision patterns in chess outcomes. The process reinforced the importance of proper data handling, balanced evaluation metrics, and regularisation to mitigate overfitting. Iteratively retraining and validating the model provided valuable insight into hyperparameter sensitivity and model stability, showing how small design changes (like learning rate decay or dropout tuning) can significantly impact performance. Overall, the experience deepened understanding of how neural architectures interpret structured and temporal game data in real-world predictive contexts.

Limitations & Future Work

A key limitation was dataset size. The original dataset (~4 GB) had to be down sampled to 95 MB due to computational constraints, Google Colab's limited RAM and CPU capacity restricted large-scale processing. As a result, the model trained on fewer games, potentially reducing diversity and representativeness. Another limitation was training time; larger datasets or deeper models would have required hours per session, which wasn't feasible within the project's timeframe.

For future work, improvements could include training on larger or higher-quality datasets, introducing draws to model real-world outcomes more faithfully, and exploring newer architectures such as Transformers or hybrid attention models to better capture long-term dependencies and positional nuances in chess games.

AI Usage

AI tools were used solely as assistive resources to enhance understanding and documentation clarity. Specifically, chatbots were consulted to clarify deep learning concepts, ensure accurate mathematical interpretation of model formulas, and assist in structuring explanatory comments within the notebook. All code implementation, experimentation, and results were independently developed and executed by me. The AI's role was purely supportive, aimed at improving conceptual precision and presentation quality without influencing the originality of the work.

References

Brownlee, J. (2019). *Deep learning for time series forecasting: Predict the future with MLPs, CNNs and LSTMs in Python*. Machine Learning Mastery.

Chollet, F. (2017). *Deep learning with Python*. Manning Publications.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR)*. https://doi.org/10.48550/arXiv.1412.6980

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research, 15*(1), 1929–1958.

Zaremba, W., Sutskever, I., & Vinyals, O. (2014). Recurrent neural network regularization. *arXiv preprint*. https://doi.org/10.48550/arXiv.1409.2329