GoalGuru

Studenti:

Nome	Matricola
Mattia Gallucci	0512116893
Choaib Goumri	0512118390

Repository: GoalGuru

Contents

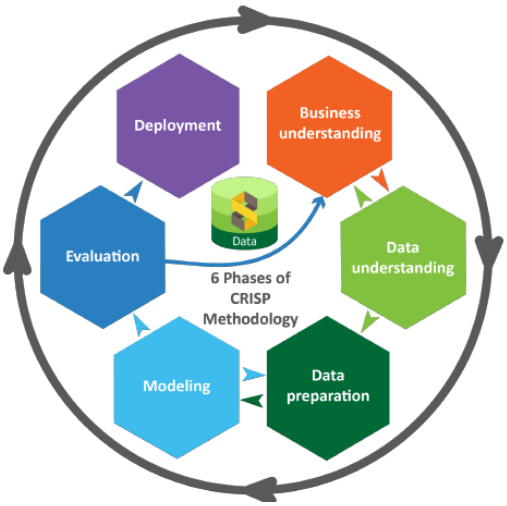
Introduzione

Per migliorare l'esperienza dei tifosi di calcio, il progetto GoalGuru utilizza una rete neurale per prevedere i risultati delle partite. Grazie a questo sistema, gli utenti possono accedere a previsioni accurate e affidabili, basate sui dati storici e sulle tendenze attuali del calcio. Questo significa che gli appassionati di calcio non dovranno più fare congetture sui risultati delle partite, ma potranno ricevere previsioni basate su analisi approfondite, rendendo la loro esperienza di visione e scommessa più informata e coinvolgente.

Inoltre, l'implementazione di un sistema di previsione basato su rete neurale può aiutare a migliorare l'engagement degli utenti, incoraggiandoli a seguire più partite e ad esplorare nuovi campionati e squadre che potrebbero aver ignorato in precedenza. Questo non solo migliora l'esperienza dei tifosi, ma può anche portare a un aumento dell'interesse e della partecipazione nel mondo del calcio.

Modello

CRISP-DM è l'acronimo di Cross-Industry Standard Process for Data Mining, un metodo di comprovata efficacia per l'esecuzione di operazioni di data mining. Come metodologia, comprende descrizioni delle tipiche fasi di un progetto e delle attività incluse in ogni fase e fornisce una spiegazione delle relazioni esistenti tra tali attività.



Business Understanding

Obiettivi

Lo scopo del progetto consiste nella realizzazione di una rete neurale per prevedere i risultati delle partite di calcio. Questo progetto si basa su un approccio di machine learning e utilizza dati storici per fare previsioni accurate. Gli obiettivi specifici includono:

- Migliorare l'accuratezza delle previsioni dei risultati delle partite di calcio.
- Fornire agli utenti previsioni affidabili che possano utilizzare per informare le loro decisioni, come scommesse o analisi sportive.
- Aumentare l'engagement degli utenti incoraggiandoli a seguire più partite e a esplorare nuovi campionati e squadre.
- Fidelizzare gli utenti offrendo un servizio di previsioni di alta qualità che soddisfi le loro esigenze e le loro aspettative.

P.E.A.S.

PEAS è l'acronimo inglese di Performance Environment Actuators Sensors. È utilizzato nello studio dell'intelligenza artificiale per raggruppare in un unico termine l'ambiente operativo.

Performance: La misura di prestazione considerata prevede la precisione delle previsioni dei risultati delle partite di calcio, cercando di avvicinarsi il più possibile ai risultati reali.

Environment: L'ambiente in cui opera l'agente è il contesto delle partite di calcio, dove ha la possibilità di accedere ai dati storici e alle statistiche delle squadre e dei giocatori. Il nostro ambiente risulta:

- Completamente osservabile: l'agente ha accesso a tutte le informazioni necessarie riguardanti le partite, le squadre e i giocatori in ogni momento.
- Stocastico: lo stato dell'ambiente cambia indipendentemente dalle previsioni fatte dall'agente.
- Episodico: la predizione per una partita non influenza direttamente quella delle partite successive.
- Statico: L'ambiente (il dataset) non cambia mentre l'agente sta elaborando le sue decisioni.
- Discreto: le previsioni dipendono dalle caratteristiche discrete delle partite considerate.
- Ad Agente Singolo: vi sarà un unico agente ad operare.

Actuators: L'agente agisce sull'ambiente fornendo previsioni sui risultati delle partite di calcio.

Sensors: L'agente percepisce le informazioni attraverso l'accesso diretto ai dati storici delle partite e alle statistiche delle squadre e dei giocatori.

Requisiti

Il sistema proposto dovrà essere in grado di:

- Raccogliere e analizzare i dati relativi alle partite di calcio, inclusi i risultati storici, le statistiche delle squadre e dei giocatori.
- Utilizzare algoritmi di machine learning per generare previsioni accurate in tempo reale, basate sui dati raccolti.

Risorse e Tool

Ci avvarremo del sito Kaggle per l'acquisizione del dataset, fondamentale per il nostro sistema di previsione. Per l'analisi, la modellazione, l'addestramento e la visualizzazione grafica dei dati e del modello verranno utilizzati diversi tool, tra i quali: Python in concomitanza di varie librerie, come pandas, numpy, matplotlib e seaborn per le informazioni sui dati, e scikit-learn per la fase di feature engineering e la fase di modeling.

Data Understanding

In questa seconda fase del CRISP-DM verrà analizzato il dataset in modo approfondito per comprendere la sua struttura, il contenuto, le relazioni tra le variabili e le eventuali problematiche o limitazioni dei dati.

Descrizione dei dati

Il dataset considerato presenta 4788 entry. Di ogni partita vengono riportate le seguenti caratteristiche:

- **date**: la data della partita.
- **time**: l'ora della partita.
- **comp**: la competizione della partita.
- **round**: il turno della partita.
- **day**: il giorno della settimana della partita.

- **venue**: il luogo della partita.
- **result**: il risultato della partita.
- **gf**: i gol segnati dalla squadra di casa.
- **ga**: i gol segnati dalla squadra ospite.
- **opponent**: l'avversario della squadra di casa.
- **xg**: gli expected goals (gol attesi) della squadra di casa.
- **xga**: gli expected goals (gol attesi) della squadra ospite.
- **poss**: il possesso palla della squadra di casa.
- **captain**: il capitano della squadra di casa.
- **formation**: la formazione della squadra di casa.
- **referee**: l'arbitro della partita.
- **sh**: i tiri della squadra di casa.
- **sot**: i tiri in porta della squadra di casa.
- **dist**: la distanza media dei tiri della squadra di casa.
- **fk**: i calci di punizione della squadra di casa.
- **pk**: i calci di rigore della squadra di casa.
- **pka**: i calci di rigore tentati dalla squadra di casa.
- **season**: l'anno della stagione della partita.
- **team**: la squadra di casa.

Esplorazione dei dati

In questa fase verranno condotte analisi esplorative per scoprire schemi, tendenze, correlazioni o anomalie nei dati delle partite di calcio. Tutte le analisi verranno condotte con l'ausilio di Python, utilizzando le librerie pandas e matplotlib per l'elaborazione e la visualizzazione dei dati.

Statistiche

Analizziamo le statistiche riguardanti il dataset.

```
print("\nStatistiche descrittive:")
print(df.describe())
```

	Unnamed:0	gf	ga	xg	xga
count	4788.000000	4788.000000	4788.000000	4788.000000	4788.000000
mean	63.044069	1.447995	1.405388	1.396512	1.364745
std	42.865191	1.312635	1.286927	0.828847	0.814947
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	28.000000	0.000000	0.000000	0.800000	0.700000
50%	62.000000	1.000000	1.000000	1.300000	1.200000
75%	87.000000	2.000000	2.000000	1.900000	1.800000
max	182.000000	9.000000	9.000000	7.000000	7.000000

	poss	attendance	notes	sh	sot
count	4788.000000	3155.000000	0.0	4788.000000	4788.000000

	poss	attendance	notes	sh	sot
mean	50.432957	38397.586688	NaN	12.619256	4.261278
std	12.810958	17595.849137	NaN	5.548444	2.459963
min	18.000000	2000.000000	NaN	0.000000	0.000000
25%	41.000000	25513.500000	NaN	9.000000	2.000000
50%	51.000000	36347.000000	NaN	12.000000	4.000000
75%	60.000000	53235.500000	NaN	16.000000	6.000000
max	82.000000	75546.000000	NaN	36.000000	15.000000

	dist	fk	pk	pkatt	season
count	4786.000000	4788.000000	4788.000000	4788.00000	4788.000000
mean	17.356247	0.453216	0.118212	0.14599	2022.365079
std	3.049341	0.665250	0.342362	0.37937	1.461850
min	5.300000	0.000000	0.000000	0.00000	2020.000000
25%	15.400000	0.000000	0.000000	0.00000	2021.000000
50%	17.200000	0.000000	0.000000	0.00000	2023.000000
75%	19.100000	1.000000	0.000000	0.00000	2024.000000
max	39.900000	4.000000	3.000000	3.00000	2024.000000

Informazioni

Ora passiamo alle informazioni del dataset e sui tipi.

```
print("\nInformazioni sul dataset:")
print(df.info())
```

#	Column	Non-Null Count	Dtype
0	Unnamed:0	4788 non-null	int64
1	date	4788 non-null	object
2	time	4788 non-null	object
3	comp	4788 non-null	object
4	round	4788 non-null	object
5	day	4788 non-null	object
6	venue	4788 non-null	object
7	result	4788 non-null	object
8	gf	4788 non-null	int64
9	ga	4788 non-null	int64
10	opponent	4788 non-null	object
11	xg	4788 non-null	float64
12	xga	4788 non-null	float64
13	poss	4788 non-null	int64
14	attendance	3155 non-null	float64
15	captain	4788 non-null	object
16	formation	4788 non-null	object
17	referee	4788 non-null	object
18	match report	4788 non-null	object
19	notes	0 non-null	float64
20	sh	4788 non-null	int64
21	sot	4788 non-null	int64
22	dist	4788 non-null	float64
23	fk	4788 non-null	int64
24	pk	4788 non-null	int64

#	Column	Non-Null Count	Dtype
25	pkatt	4788 non-null	int64
26	season	4788 non-null	int64
27	team	4788 non-null	object
dtypes:	float64(5),	int64(10),	object(13)

Notiamo che l'entry **notes** contiene solo valori **null**, mentre **attendance** presenta alcuni valori **null**.

Valori duplicati

Eseguendo

```
print("\nNumero di duplicati nel dataset:", df.duplicated().sum())
```

notiamo che non sono presenti duplicati nel dataset.

Data Preparation

Dopo aver acquisito e analizzato i dati, nella fase di Data Preparation questi vengono preparati affinché possano essere utilizzati in fase di addestramento dell'algoritmo di Machine Learning. In questa fase risulta utile l'uso della libreria sklearn, specialmente per il processo di Feature Engineering, come l'One-Hot Encoding e la standardizzazione delle variabili numeriche.

Data cleaning

Rimozione colonne irrilevanti

Rimosse colonne: **Unnamed: 0**, **comp**, **round**, **attendance**, **match report**, **notes**.

```
df.drop(columns=["Unnamed: 0", "comp", "round", "attendance", "match report", "notes"],
    inplace=True)
```

Conversione del tipo dei dati

date: convertito in formato datetime.

venue, **opponent**, **team**, **result**: convertiti in tipo categoria.

```
df["date"] = pd.to_datetime(df["date"])
df['venue'] = df['venue'].astype('category')
df['opponent'] = df['opponent'].astype('category')
df['team'] = df['team'].astype('category')
df['result'] = df['result'].astype('category')
```

Estrazione di informazioni temporali

Aggiunta colonna **day** (nome del giorno) da **date**.

Creata colonna **hour** (ora del match) da **time**.

Creata colonna **day_code** (giorno della settimana come numero) da **date**.

```
df['day'] = df['date'].dt.day_name()
df["hour"] = df["time"].str.replace(":.+", "", regex=True).astype("int")
df["day_code"] = df["date"].dt.dayofweek
```

Pulizia e consolidamento

Rimosse stringhe extra da **formation** (es. "", "-0").

Consolidate formazioni poco frequenti in una categoria unica **Altro**.

```
df.formation = df.formation.str.replace("   ", "")
df.formation = df.formation.str.replace("-0", "")
value_counts = df.formation.value_counts()
to_replace = value_counts[value_counts < 107].index
df['formation'] = df['formation'].replace(to_replace, 'Altro')
```

Calcolo dei punti e vincitori della stagione

Assegnazione dei punti in base al risultato (W = 3, D = 1, L = 0).

Calcolo dei vincitori di ogni stagione e aggiunta della colonna **season_winner**.

```
df['points'] = df['result'].apply(lambda x: 3 if x == 'W' else 1 if x == 'D' else 0)
df['points'] = df['points'].astype('int')
winners = df.groupby(['season', 'team'], observed=False)['points'].sum().reset_index() \
  .sort_values(['season', 'points'], ascending=[True, False]) \
  .groupby('season', observed=False).first()
df['season_winner'] = df['season'].map(winners['team'])
```

Gestione valori nulli

Gestione dei dati mancanti in **captain** con sostituzione o rimozione.

```
def captains_func(data):
    if data['count'] == 0:
        data['count'] = np.nan
    return data
group = df.groupby('team', observed=False)['captain'].value_counts().reset_index(name='
    count')
group = group.apply(captains_func, axis=1)
group.dropna(inplace=True)
group = group.drop(columns='count')
```

Feature scaling

Creazione di rapporti numerici

fk_ratio: rapporto tra calci di punizione (**fk**) e tiri (**sh**).

pk_conversion_rate: rapporto tra rigori segnati (**pk**) e tentati (**pkatt**).

pk_per_shot: rapporto tra rigori tentati (**pkatt**) e tiri (**sh**).

```
def calculate_fk_pk_ratios(data):
    data['fk_ratio'] = data['fk'] / data['sh']
    data['pk_conversion_rate'] = data['pk'] / data['pkatt']
    data['pk_per_shot'] = data['pkatt'] / data['sh']
    # Gestione dei valori infiniti
    data['fk_ratio'] = data['fk_ratio'].replace([np.inf, -np.inf], np.nan)
    data['pk_conversion_rate'] = data['pk_conversion_rate'].replace([np.inf, -np.inf],
    np.nan)
    data['pk_per_shot'] = data['pk_per_shot'].replace([np.inf, -np.inf], np.nan)
    # Conversione in percentuali
    data['fk_percentage'] = data['fk_ratio'] * 100
    data['pk_conversion_percentage'] = data['pk_conversion_rate'] * 100
    data['pk_per_shot_percentage'] = data['pk_per_shot'] * 100
    return data
# Applicazione della funzione
df_sorted = calculate_fk_pk_ratios(df_sorted)
# Rimozione delle colonne non necessarie
df_sorted.drop(['pk_conversion_rate', 'pk_conversion_percentage'], axis=1, inplace=True)
```

Standardizzazione/Normalizzazione:

Utilizzo delle medie mobili per calcolare valori scalati nel tempo (**rolling_xg**, **rolling_xga**, etc.).

Applicazione One-Hot Encoding alle variabili categoriche e StandardScaler per le variabili numeriche.

```
def calculate_rolling_average(data, column, window=5):
    return data.groupby('team', observed=False)[column].transform(
        lambda x: x.rolling(window=window, min_periods=1).mean()
    )
# Applicazione della funzione alle colonne selezionate
df_sorted['rolling_xg'] = calculate_rolling_average(df_sorted, 'xg')
df_sorted['rolling_xga'] = calculate_rolling_average(df_sorted, 'xga')
df_sorted['rolling_poss'] = calculate_rolling_average(df_sorted, 'poss')
df_sorted['rolling_sh'] = calculate_rolling_average(df_sorted, 'sh')
df_sorted['rolling_sot'] = calculate_rolling_average(df_sorted, 'sot')
df_sorted['rolling_dist'] = calculate_rolling_average(df_sorted, 'dist')
# Codifica del risultato in valori numerici (W = 1, D = 0, L = -1)
df_sorted['result_encoded'] = pd.to_numeric(df_sorted['result'].map({'W': 1, 'D': 0, 'L'
    : -1}))
# Calcolo della forma (media mobile dei risultati)
df_sorted['form'] = calculate_rolling_average(df_sorted, 'result_encoded')

# One-Hot Encoding per le variabili categoriche
X_categorical_encoded = pd.get_dummies(X[categorical_cols], columns=categorical_cols)
# Standardizzazione delle variabili numeriche
scaler = StandardScaler()
X_numerical_scaled = pd.DataFrame(scaler.fit_transform(X[numerical_cols]), columns=
    numerical_cols)
```

Feature selection

Rimozione colonne irrilevanti al modello

Eliminati campi che non aggiungono valore predittivo diretto: **gf**, **ga**, **xg**, **xga**, **poss**, **points**, etc.

```
columns_to_drop = ['gf', 'ga', 'xg', 'xga', 'poss', 'sh', 'sot',
                   'goal_diff', 'day', 'pk', 'pkatt', 'fk',
                   'referee', 'dist','points', 'season_winner', 'hour', 'result_encoded'
    , 'day_code']
df_sorted = df_sorted.drop(columns=columns_to_drop)
```

Data Modeling

Nella fase di Data Modeling verranno determinate e valutate le tecniche finalizzate alla costruzione del modello, dopodiché ed si passerà alla fase di addestramento di quest'ultimo, durante la quale verranno configurati i parametri, verrà addestrato e si commenteranno i risultati ottenuti. Il problema affrontato è di apprendimento non supervisionato.

Suddivisione del Dataset

In questa fase, il dataset viene preparato separando le variabili predittive (X) dall'etichetta di target (y), e poi suddividendo i dati in un set di **training** e uno di **validation**. Questo processo è essenziale per addestrare e valutare il modello in modo efficace, evitando il rischio di overfitting e garantendo che il modello generalizzi bene sui dati non visti.

```
X = tr.drop('result', axis=1)
y = tr['result']
# Unione delle v# Codifica del target (y) in valori numerici
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
# Creazione del test set (ultima stagione)
test_x = X_final.tail(761)
```

```
test_y = y_encoded[-761:]
test_x['result'] = test_y
test_x.to_csv("test_set.csv", index=False)
# Rimozione dell'ultima stagione dal training set
X_final = X_final.iloc[:-761]
y_encoded = y_encoded[:-761]
# Divisione in training e validation set
X_train, X_val, y_train, y_val = train_test_split(X_final, y_encoded, test_size=0.2,
    random_state=42)
```

Creazione del Modello di Rete Neurale

La rete neurale è composta da **due strati densi (dense layers)** e da un **layer di dropout**. Ogni strato denso è completamente connesso, il che significa che ogni neurone in uno strato è connesso a tutti i neuroni dello strato successivo.

- **Strato di input**: L'input del modello è una serie di variabili numeriche e categoriche. La dimensione dell'input è determinata dal numero di variabili presenti nel dataset.
- **Strati nascosti (Hidden Layers)**:
  - **Primo strato nascosto**: Il primo strato è una **layer denso** che contiene **neurons_1layer** neuroni, dove il numero di neuroni viene scelto tramite iperparametri.
  - **Secondo strato nascosto**: Il secondo strato è anch'esso un **layer denso** che contiene **neurons_2layer** neuroni. Anche in questo caso, il numero di neuroni è selezionato tramite iperparametri.
- **Dropout**: Un layer di **dropout** con una probabilità di 0.1 viene applicato dopo il secondo strato nascosto. Questo serve a **prevenire l'overfitting**, in quanto disattiva casualmente il 10% dei neuroni durante ogni aggiornamento del peso, forzando la rete a generalizzare meglio.
- **Strato di output**: Lo strato finale è un **layer denso** con 3 neuroni, uno per ciascuna delle classi: **vittoria**, **pareggio** e **sconfitta**. L'output è generato usando la funzione di attivazione **softmax**, che restituisce le probabilità per ciascuna classe, dove la somma delle probabilità è pari a 1.

```
def create_network(input_dim, neurons_1layer, neurons_2layer, activation_function):
    inputs = tf.keras.Input((input_dim,))
    x = layers.Dense(neurons_1layer, activation_function)(inputs)
    x = layers.Dense(neurons_2layer, activation_function)(x)
    x = layers.Dropout(0.1)(x)
    output = layers.Dense(3, "softmax")(x)
    model = tf.keras.Model(inputs=inputs, outputs=output, name="neural_net")
    return model
```

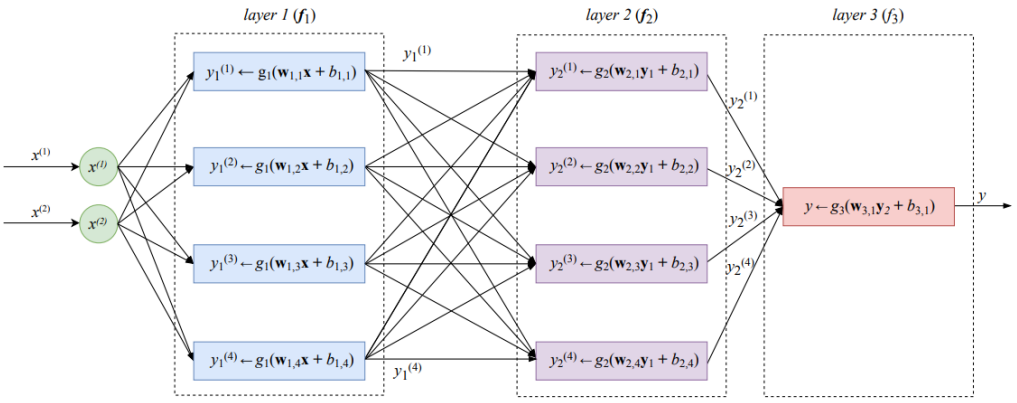


Immagine a scopo illustrativo

Funzione di Attivazione

La rete utilizza una **funzione di attivazione** non lineare, che è una componente chiave per introdurre flessibilità nel modello. Le funzioni di attivazione disponibili nel codice sono:

- **ReLU** (**relu**): Funzione di attivazione comunemente utilizzata nelle reti neurali per la sua capacità di evitare il problema del gradiente che scompare e di introdurre non linearità.
- **Sigmoid** (**sigmoid**): Funzione di attivazione che restituisce un valore tra 0 e 1, utile quando l'output deve essere interpretato come una probabilità.
- **Tanh** (**tanh**): Funzione di attivazione simile al Sigmoid, ma con un output che varia tra -1 e 1.

Funzione di Perdita

La **funzione di perdita** utilizzata è **sparse_categorical_crossentropy**, che è adatta per la classificazione multiclasse. In questo caso, le tre possibili etichette (vittoria, pareggio, sconfitta) sono trattate come classi distinte, e la funzione di perdita misura l'entropia incrociata tra le probabilità predette e quelle reali.

```
model.compile(
        loss='sparse_categorical_crossentropy',
        optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
        metrics=['accuracy']
    )
```

Ricerca degli Iperparametri

Grid Search La ricerca degli iperparametri è un processo fondamentale per ottimizzare le prestazioni di un modello. In questo caso, viene utilizzato il **grid search** per esplorare diverse combinazioni di iperparametri, come il numero di neuroni negli strati nascosti, la funzione di attivazione, il learning rate, il numero di epoche, e la dimensione del batch. Ogni combinazione viene testata e il modello con la **minore perdita di validazione** viene selezionato.

```
# Griglia di iperparametri
GRID_SEARCH = {
    "learning_rate": [1e-3],
    "epochs": [5, 6, 7, 8, 9, 10],
    "neurons_1layer": [50, 55],
    "neurons_2layer": [30, 50],
    "activation_functions": ['relu', 'sigmoid', 'tanh'],
    "batch_size": [200]
}

# Creazione di tutte le combinazioni di iperparametri
grid_combinations = list(itertools.product(
    GRID_SEARCH['learning_rate'],
    GRID_SEARCH['epochs'],
    GRID_SEARCH['neurons_1layer'],
    GRID_SEARCH['neurons_2layer'],
    GRID_SEARCH['activation_functions'],
    GRID_SEARCH['batch_size']
))

# Variabili per tenere traccia dei migliori iperparametri
best_params = None
best_val_loss = np.inf

# Ciclo per testare ogni combinazione di iperparametri
for combination in grid_combinations:
    learning_rate, epochs, neurons_1layer, neurons_2layer, activation_function,
    batch_size = combination

    print(f"Testing combination: lr={learning_rate}, epochs={epochs}, neurons_1layer={
    neurons_1layer}, neurons_2layer={neurons_2layer}, activation={activation_function},
    batch_size={batch_size}")

    # Creazione del modello con i parametri correnti
```

```
    model = create_network(X_train.shape[1], neurons_1layer, neurons_2layer,
    activation_function)

    # Compilazione del modello
    model.compile(
        loss='sparse_categorical_crossentropy',
        optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
        metrics=['accuracy']
    )

    # Addestramento del modello
    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=epochs,
        batch_size=batch_size,
        verbose=0  # Nessun output durante la ricerca
    )

    # Valutazione della loss sul validation set
    final_val_loss = history.history['val_loss'][-1]

    print(f"Validation loss: {final_val_loss}")

    # Aggiornamento dei migliori iperparametri
    if final_val_loss < best_val_loss:
        best_val_loss = final_val_loss
        best_params = {
            "learning_rate": learning_rate,
            "epochs": epochs,
            "neurons_1layer": neurons_1layer,
            "neurons_2layer": neurons_2layer,
            "activation_function": activation_function,
            "batch_size": batch_size
        }

# Stampa dei migliori iperparametri
print("Best hyperparameters found:")
print(best_params)
print(f"Best validation loss: {best_val_loss}")
```

Iperparametri Testati

Durante il **grid search**, vengono esplorati i seguenti iperparametri:

- **learning_rate**: La velocità con cui l'ottimizzatore aggiorna i pesi. Un valore troppo alto può causare instabilità, mentre uno troppo basso può rallentare l'apprendimento.
- **epochs**: Il numero di epoche rappresenta il numero di volte in cui il modello vedrà l'intero dataset durante l'addestramento. Maggiore è il numero di epoche, maggiore è la possibilità di apprendimento, ma rischia anche di causare overfitting.
- **neurons_1layer** e **neurons_2layer**: Il numero di neuroni nel primo e secondo strato della rete. Un numero maggiore di neuroni permette una maggiore capacità di apprendimento, ma può anche portare a un modello più complesso e incline all'overfitting.
- **activation_functions**: Le funzioni di attivazione da testare. Ogni funzione può influire sul modo in cui la rete apprende e sulle prestazioni finali.
- **batch_size**: La dimensione del batch, che determina quante istanze di dati vengono utilizzate per ogni aggiornamento dei pesi. Un valore maggiore di batch size riduce il rumore ma può richiedere più risorse computazionali.

Selezione del Modello Migliore

Alla fine del grid search, il modello con il **minor valore di perdita di validazione** (validation loss) viene selezionato. I parametri di questo modello migliore vengono poi utilizzati per allenare il modello finale.

Random Search Ora viene utilizzato il **random search** per esplorare combinazioni casuali di iperparametri, come il numero di neuroni negli strati nascosti, la funzione di attivazione, il learning rate, il numero di epoche, e la dimensione del batch. A differenza del grid search, il random search permette di testare un numero ridotto di combinazioni, riducendo il tempo necessario per la ricerca senza compromettere significativamente le possibilità di trovare iperparametri ottimali.

```
# Definizione dello spazio degli iperparametri per Random Search
RANDOM_SEARCH_SPACE = {
    "learning_rate": [1e-3, 1e-4, 1e-5],
    "epochs": [5, 10, 15, 20],
    "neurons_1layer": [30, 50, 70],
    "neurons_2layer": [20, 40, 60],
    "activation_functions": ['relu', 'sigmoid', 'tanh'],
    "batch_size": [100, 200, 300]
}

# Numero di combinazioni casuali da provare
NUM_RANDOM_COMBINATIONS = 20

# Variabili per tenere traccia dei migliori iperparametri
best_params = None
best_val_loss = np.inf

# Ciclo per testare combinazioni casuali di iperparametri
for _ in range(NUM_RANDOM_COMBINATIONS):
    # Selezione casuale degli iperparametri
    learning_rate = random.choice(RANDOM_SEARCH_SPACE["learning_rate"])
    epochs = random.choice(RANDOM_SEARCH_SPACE["epochs"])
    neurons_1layer = random.choice(RANDOM_SEARCH_SPACE["neurons_1layer"])
    neurons_2layer = random.choice(RANDOM_SEARCH_SPACE["neurons_2layer"])
    activation_function = random.choice(RANDOM_SEARCH_SPACE["activation_functions"])
    batch_size = random.choice(RANDOM_SEARCH_SPACE["batch_size"])

    print(f"Testing combination: lr={learning_rate}, epochs={epochs}, neurons_1layer={
    neurons_1layer}, neurons_2layer={neurons_2layer}, activation={activation_function},
    batch_size={batch_size}")

    # Creazione del modello con i parametri correnti
    model = create_network(X_train.shape[1], neurons_1layer, neurons_2layer,
    activation_function)

    # Compilazione del modello
    model.compile(
        loss='sparse_categorical_crossentropy',
        optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
        metrics=['accuracy']
    )

    # Addestramento del modello
    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=epochs,
        batch_size=batch_size,
        verbose=0  # Nessun output durante la ricerca
    )

    # Valutazione della loss sul validation set
    final_val_loss = history.history['val_loss'][-1]

    print(f"Validation loss: {final_val_loss}")

    # Aggiornamento dei migliori iperparametri
    if final_val_loss < best_val_loss:
        best_val_loss = final_val_loss
        best_params = {
            "learning_rate": learning_rate,
            "epochs": epochs,
            "neurons_1layer": neurons_1layer,
            "neurons_2layer": neurons_2layer,
```

```
            "activation_function": activation_function,
            "batch_size": batch_size
        }

# Stampa dei migliori iperparametri
print("Best hyperparameters found:")
print(best_params)
print(f"Best validation loss: {best_val_loss}")
```

Iperparametri Testati
Durante il **random search**, vengono esplorati i seguenti iperparametri:

- **learning_rate**: La velocità con cui l'ottimizzatore aggiorna i pesi. Un valore troppo alto può causare instabilità, mentre uno troppo basso può rallentare l'apprendimento.
- **epochs**: Il numero di epoche rappresenta il numero di volte in cui il modello vedrà l'intero dataset durante l'addestramento. Maggiore è il numero di epoche, maggiore è la possibilità di apprendimento, ma rischia anche di causare overfitting.
- **neurons_1layer e neurons_2layer**: Il numero di neuroni nel primo e secondo strato della rete. Un numero maggiore di neuroni permette una maggiore capacità di apprendimento, ma può anche portare a un modello più complesso e incline all'overfitting.
- **activation_function**: Le funzioni di attivazione da testare. Ogni funzione può influire sul modo in cui la rete apprende e sulle prestazioni finali.
- **batch_size**: La dimensione del batch, che determina quante istanze di dati vengono utilizzate per ogni aggiornamento dei pesi. Un valore maggiore di batch size riduce il rumore ma può richiedere più risorse computazionali.

Selezione del Modello Migliore
Durante il random search, vengono generate combinazioni casuali di iperparametri e, per ognuna di esse, viene addestrato un modello. La perdita di validazione (validation loss) viene utilizzata per determinare la qualità del modello. Il modello con la minore perdita di validazione viene selezionato come il migliore. Gli iperparametri di questo modello vengono poi utilizzati per allenare il modello finale.

K-Nearest Neighbors (KNN)

Il KNN è un algoritmo di apprendimento supervisionato semplice ma potente, utilizzato principalmente per problemi di classificazione e regressione. Il principio fondamentale del KNN è che oggetti simili si trovano vicini nello spazio delle caratteristiche.

Input del Modello
Il modello riceve in input una serie di variabili numeriche e categoriali, rappresentate come vettori nello spazio multidimensionale. Questi vettori definiscono la posizione di ciascun punto (osservazione) nel dataset.

Parametri Chiave del Modello

Numero di Vicini (k): Indica quanti punti più vicini al dato di input devono essere considerati per effettuare una classificazione. Un valore basso di kkk può rendere il modello troppo sensibile ai rumori, mentre un valore alto può rendere il modello troppo generico.

Peso dei Vicini (weights): Determina l'importanza dei vicini nella decisione finale. Può essere uniforme (tutti i vicini hanno lo stesso peso) o basato sulla distanza (i vicini più vicini hanno un peso maggiore).

Metrica di Distanza (metric): Specifica il metodo per calcolare la distanza tra i punti, ad esempio la distanza euclidea o manhattan.

```
GRID_SEARCH = {
    "n_neighbors": [3, 5, 7, 9, 11],  # Numero di vicini
    "weights": ['uniform', 'distance'],  # Peso dei vicini
    "metric": ['euclidean', 'manhattan']  # Metrica di distanza
}

```

```
# Creazione di tutte le combinazioni di iperparametri
grid_combinations = list(itertools.product(
    GRID_SEARCH['n_neighbors'],
    GRID_SEARCH['weights'],
    GRID_SEARCH['metric']
))
```

Funzionamento del Modello

Durante la fase di predizione, il modello cerca i k punti più vicini all'osservazione di input basandosi sulla metrica di distanza scelta.

Il modello assegna la classe predominante tra i k vicini per classificare l'osservazione, oppure calcola una media ponderata per i problemi di regressione.

```
for combination in grid_combinations:
    n_neighbors, weights, metric = combination

    print(f"Testing combination: n_neighbors={n_neighbors}, weights={weights}, metric={
    metric}")

    # Creazione del modello KNN con i parametri correnti
    model = KNeighborsClassifier(n_neighbors=n_neighbors, weights=weights, metric=metric
    )

    # Addestramento del modello
    model.fit(X_train, y_train)

    # Valutazione dell'accuratezza sul validation set
    val_accuracy = model.score(X_val, y_val)

    # Aggiungi l'accuratezza alla lista
    val_accuracies.append(val_accuracy)

    print(f"Validation accuracy: {val_accuracy}")

    # Aggiornamento dei migliori iperparametri
    if val_accuracy > best_val_accuracy:
        best_val_accuracy = val_accuracy
        best_params = {
            "n_neighbors": n_neighbors,
            "weights": weights,
            "metric": metric
        }

# Stampa dei migliori iperparametri
print("Best hyperparameters found:")
print(best_params)
print(f"Best validation accuracy: {best_val_accuracy}")
```

Iperparametri Testati
Il processo di ottimizzazione degli iperparametri prevede la selezione delle migliori combinazioni tra:

- Numero di vicini (kkk): Valori testati tra 3 e 11.
- Peso dei vicini: Uniforme o basato sulla distanza.
- Metrica di distanza: Euclidea o manhattan.

Selezione del Modello Migliore
Tutte le combinazioni di iperparametri vengono testate utilizzando un validation set. L'accuratezza sul validation set è la metrica utilizzata per valutare ciascun modello. La combinazione di iperparametri con la migliore accuratezza viene selezionata per creare il modello finale.

Valutazione del modello

La rete neurale è stata scelta per la sua capacità di modellare relazioni non lineari e complesse tra le variabili, essenziali per il dataset calcistico, e per la sua flessibilità nel gestire problemi multiclasse tramite lo strato di output con attivazione softmax. L'ottimizzazione del modello è stata realizzata tramite Grid Search, che ha esplorato combinazioni di iperparametri chiave come numero di neuroni, funzioni di attivazione e tasso di dropout, garantendo la selezione della configurazione migliore.

Questo approccio sistematico ha permesso di bilanciare la complessità del modello con la capacità di generalizzazione, riducendo il rischio di overfitting e migliorando le prestazioni complessive.

Dopo aver addestrato il modello utilizzando il set di addestramento, il passo successivo è la **valutazione delle prestazioni** del modello sui dati di validazione. Questo passaggio è cruciale per capire quanto il modello generalizzi bene su dati che non ha mai visto durante l'addestramento. La valutazione si concentra su diverse metriche per determinare se il modello sta facendo previsioni accurate e affidabili.

Valutazione sulla Base dei Dati di Validazione

Una volta che il modello è stato addestrato, utilizziamo i dati di **validazione** per misurare quanto bene il modello si adatti ai dati non visti. Questo viene fatto utilizzando la funzione **evaluate**() di Keras:

```
val_loss, val_accuracy = best_model.evaluate(X_val, y_val)
```

- **Perdita di validazione (val_loss)**: Indica quanto il modello si discosta dalle etichette reali nei dati di validazione. Più bassa è la perdita, migliore è il modello. Una perdita elevata suggerisce che il modello non sta performando bene.
- **Precisione di validazione (val_accuracy)**: Indica la percentuale di previsioni corrette sui dati di validazione. Una precisione elevata indica che il modello sta facendo previsioni accurate.

Grafico della Perdita durante l'Addestramento e la Validazione

Durante l'addestramento, è utile visualizzare come la **perdita** del modello cambia nel tempo, sia per i dati di addestramento che per quelli di validazione. Questo aiuta a identificare fenomeni come l'**overfitting** (quando il modello si adatta troppo ai dati di addestramento, ma performa male sui dati di validazione) o l'**underfitting** (quando il modello non si adatta abbastanza ai dati).

```
plt.figure(figsize=(8, 6))
plt.plot(best_history.history['loss'], label='Perdita di addestramento')
plt.plot(best_history.history['val_loss'], label='Perdita di validazione')
plt.title('Andamento della Perdita')
plt.xlabel('Epoca')
plt.ylabel('Perdita')
plt.legend()
plt.show()
```

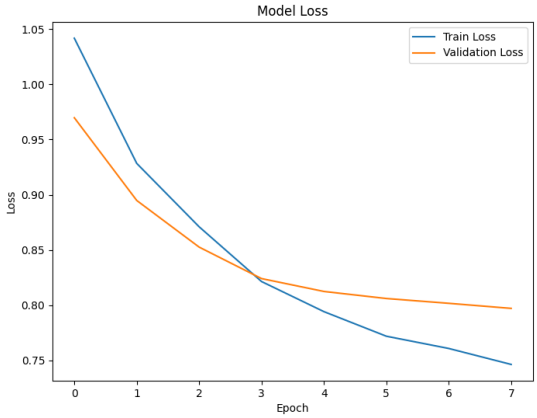
- Se la **perdita di validazione** continua a crescere mentre la **perdita di addestramento** diminuisce, ciò suggerisce un overfitting.
- Se entrambe le perdite sono alte, il modello potrebbe non essere abbastanza complesso (underfitting).

Matrice di Confusione

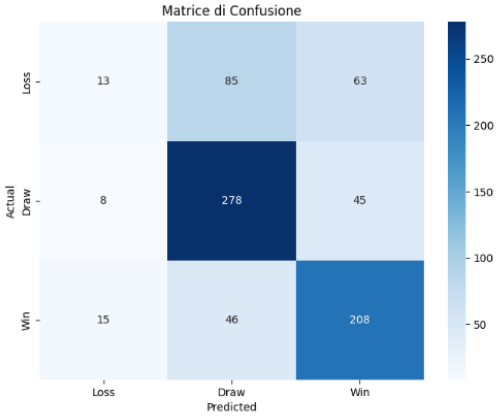
La **matrice di confusione** è una tabella che visualizza il numero di previsioni corrette e errate per ogni classe. Aiuta a capire meglio come il modello si comporta nei confronti di ciascuna delle classi. Con una matrice di confusione, possiamo vedere se il modello sta facendo confusione tra alcune classi.

```
# Calcolo della matrice di confusione
conf_matrix = confusion_matrix(ts_y, y_pred_classes)

# Visualizzazione della matrice di confusione
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Loss', 'Draw', 'Win'],
```




```
            yticklabels=['Loss', 'Draw', 'Win'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Matrice di Confusione')
plt.show()
```



Conclusioni

Migliori Iperparametri Trovati

Dopo aver eseguito una ricerca sugli iperparametri, i migliori valori ottenuti per il modello sono:

- **Learning Rate**: 0.001
- **Epochs**: 8
- **Numero di Neuroni (Layer 1)**: 55
- **Numero di Neuroni (Layer 2)**: 50
- **Funzione di Attivazione**: ReLU
- **Batch Size**: 200

Questi iperparametri sono stati scelti dopo aver esplorato diverse combinazioni e ottimizzato il modello per ottenere le migliori prestazioni sul dataset.

Riepilogo del Modello

Il modello caricato (**best_model.h5**) ha la seguente architettura:

- **Input Layer**: 273 unità in ingresso, corrispondenti alle variabili predittive.
- **Primo Strato Denso (Dense)**: 55 neuroni, con 15,070 parametri addestrabili.
- **Secondo Strato Denso (Dense)**: 50 neuroni, con 2,800 parametri addestrabili.
- **Dropout**: Applicato al secondo strato denso con un tasso di dropout che impedisce l'overfitting.
- **Output Layer (Dense)**: 3 neuroni in uscita per le 3 classi da predire (vittoria, pareggio, sconfitta), con 153 parametri addestrabili.

In totale, il modello ha **18,025 parametri** (di cui 18,023 sono addestrabili), con una dimensione di **70.41 KB.**

Valutazione sul Test Set

Il modello è stato valutato sui dati di test (**ts_x**, **ts_y**) e i risultati ottenuti sono i seguenti:

- **Test Loss**: 78.11%
- **Test Accuracy**: 67.15%

Questi valori indicano che, sebbene il modello mostri una **precisione del 67%**, la **perdita** (che misura l'errore complessivo) è relativamente alta. Ciò suggerisce che il modello potrebbe non essere perfettamente ottimizzato e che potrebbe essere migliorato con ulteriori tentativi di tuning degli iperparametri o con tecniche di regolarizzazione.

Report di Classificazione

Il modello è stato testato anche per la **classificazione** delle tre classi (vittoria, pareggio, sconfitta). Ecco i risultati del report di classificazione:

	precision	**recall**	**f1-score**	**support**
0	0.47	0.14	0.21	161
1	0.68	0.84	0.75	331
2	0.70	0.78	0.74	269
accuracy			0.67	761
macro avg	0.61	0.59	0.57	761
weighted avg	0.64	0.67	0.63	761

Analisi dei Risultati:

- La **precisione** per la classe 0 (pareggio) è relativamente bassa (0.47), suggerendo che il modello fatica a distinguere correttamente questa classe rispetto alle altre. La bassa **recall** per la classe 0 (0.14) indica che il modello tende a **mancare molte istanze di pareggio**, probabilmente classificandole erroneamente come vittoria o sconfitta.
- La classe 1 (vittoria) ha una **precisione** di 0.68 e un **recall** di 0.84, con un **f1-score** di 0.75, che sono molto buoni. Questo significa che il modello è più efficace nell'identificare le vittorie e fa pochi falsi negativi in questa classe.
- La classe 2 (sconfitta) ha una **precisione** di 0.70 e un **recall** di 0.78, con un **f1-score** di 0.74, suggerendo un buon equilibrio tra precisione e recall per questa classe.

Conclusioni

- **Performance Complessiva**: Il modello ha un'accuratezza complessiva del 67%, che indica una prestazione decente, ma non ottimale. La classe di **pareggio** è particolarmente difficile da prevedere, e potrebbe essere utile indagare ulteriormente su come migliorare il trattamento di questa classe.
- **Disparità tra le Classi**: Il modello sembra essere migliore nel classificare vittorie e sconfitte, mentre ha difficoltà con il pareggio. Questo potrebbe essere dovuto a uno sbilanciamento nelle classi o alla difficoltà intrinseca nel modello di distinguere le situazioni di pareggio.
- **Possibili Miglioramenti**: Potrebbero essere esplorati ulteriori approcci per migliorare la classificazione del pareggio, come l'uso di tecniche di **pesatura delle classi**, l'**oversampling** delle classi sottorappresentate o l'introduzione di **nuove caratteristiche** per il modello. Inoltre, si potrebbero fare ulteriori esperimenti con **epoche più alte** o **tassi di apprendimento** diversi.

In generale, il modello ha buone basi per essere migliorato, ma presenta alcune aree che richiedono attenzione per massimizzare le sue prestazioni.