

ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA
SEDE DI CESENA
FACOLTA' DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA IN SCIENZE DELL'INFORMAZIONE

**ANALISI E STUDIO DI STRATEGIE DI GIOCO
MULTIPLAYER CON TECNICHE DI MACHINE
LEARNING**

Tesi di Laurea in
Algoritmi e strutture dati

Relatore
Chiar.mo Prof. Vittorio Maniezzo

Co-Relatore
Dott. Matteo Roffilli

Presentata da
Giovanni Lughi

Sessione III
Anno Accademico 2004/2005

*Alla mia Famiglia che crede in me,
e a Gloria per quello che mi sta dando.*

INDICE

1. Introduzione	3
1.1 AI nei giochi	3
1.2 BZFlag	6
2. Support Vector Machine	9
2.1 Introduzione	9
2.2 Fondamenti teorici	10
2.2.1 Dimensione VC	11
2.2.2 Minimizzazione dell'errore teorico	11
2.3 Classificatore lineare	12
2.3.1 Il caso linearmente separabile	12
2.3.2 Il caso non linearmente separabile	13
2.4 Macchine non lineari	14
3. LIBSVM	17
3.1 Caratteristiche generali	17
3.2 Formato dei dati	19
3.3 Normalizzazione	20
3.4 Addestramento e Test	21

INDICE

4. Interfaccia	23
4.1 Introduzione	23
4.2 Attivazione Opzioni	25
4.3 Salvataggio Dati Tank	26
4.4 Salvataggio Azioni	26
4.5 Lettura ed esecuzione Azioni	27
4.6 Salvataggio Dati Train	28
4.7 Controllo Nemici Manuale	29
4.8 Controllo Nemici Automatico	31
4.9 Salvataggio Mappa	35
5. Machine Learning in BZFlag	41
5.1 Thread	41
5.2 Mappa	42
5.3 Memorizzare le azioni	45
5.4 Eseguire le azioni	47
5.5 Data Tank	48
5.6 Controllo nemici	52
5.6.1 Training	52
5.6.2 Predict	55
5.6.3 Test e risultati	55
Conclusioni e sviluppi futuri	57
Bibliografia	59

Elenco delle figure

Fig. 1.1 - Mappa di BZF con texture Lego	8
Fig. 1.2 - Mappa di BZF con texture	8
Fig. 2.1 – La dimensione VC	11
Fig. 2.2 - Iperpiano separatore per il caso linearmente separabile	13
Fig. 2.3 - Iperpiano separatore per il caso non linearmente separabile	14
Fig. 2.4 - Mapping in uno spazio a dimensione superiore	15
Fig. 4.1 – Esempio di quattro esecuzioni del Controllo Nemici Manuale	31
Fig. 4.2 - Controllo Nemici Automatico	33
Fig. 4.3 - Controllo Nemici Automatico	33
Fig. 4.4 - Controllo Nemici Automatico	34
Fig. 4.5 - Controllo Nemici Automatico	34
Fig. 4.6 - Esempio di raffigurazione matriciale di un ostacolo	36
Fig. 4.7 - Esempio di mappa vista sul radar	38
Fig. 4.8 - Raffigurazione (rimpicciolita) su file della mappa in Fig. 4.7	38
Fig. 5.1 – Esempio di mappa con texture	44
Fig. 5.2 - Esempio di mappa con texture	44
Fig. 5.3 – Alcuni casi di calcolo della “differenza di puntamento”	50
Fig. 5.4 – Dati forniti dalla LIBSM circa l’addestramento appena eseguito	54

Elenco delle tabelle

Tab. 2.1 - Funzioni Kernel	15
Tab. 3.1 – Classe <i>svm</i> della LIBSVM	18
Tab. 3.2 – Formato dei dati per la LIBSVM	19
Tab. 4.1 – Funzione <i>doEvent</i>	24
Tab. 4.2 – Forma dei nomi dei file di input	27
Tab. 4.3 – Struttura dei dati usati dalla LIBSVM	29
Tab. 4.4 – Metodo <i>CeckEnemy</i>	30
Tab. 4.5 – Funzione <i>render</i>	32
Tab. 4.6 – Classe <i>NetworkMapper</i>	35
Tab. 4.7 – Classe <i>NetworkInterface</i>	39
Tab. 5.1 – Struttura <i>azione</i>	45
Tab. 5.2 – Esempio di output delle azioni su file	46
Tab. 5.3 – Formato dati dei file di input delle azioni	48
Tab. 5.4 – Struttura <i>dataTank</i>	48
Tab. 5.5 – Esempio di output su file dei dati dei tank in gioco	51
Tab. 5.6 – Esempio di dati di addestramento per la LIBSVM	53
Tab. 5.7 – I parametri usati nell’addestramento dell’agente con LIBSVM	53
Tab. 5.8 – Esempio di file “.model”	54

Tra tutti i tipi di applicazioni informatiche il campo dei videogiochi è tra quelli di più diffuso consumo e maggior redditività; per questo è anche uno dei settori più avanzati ed in cui maggiormente ci si occupa di sperimentazione ed innovazione.

Al giorno d'oggi due sono le frontiere ancora aperte in questo mondo: la realtà virtuale e l'intelligenza artificiale.

In particolare questa tesi di laurea si occupa di intelligenza artificiale e di una delle sue branche meno esplorate nel campo videoludico, il Machine Learning.

È nostra opinione che sfruttando queste tecniche di apprendimento sia possibile ottenere grandi risultati nel miglioramento delle intelligenze artificiali, sia nell'industria dei videogiochi che in molte altre applicazioni informatiche.

Il progetto BZFlag da noi iniziato si prefissa appunto di studiare l'applicabilità del Machine Learning ad una particolare categoria di videogiochi e di dimostrare la bontà dei risultati ottenibili.

È stato scelto il videogioco BZFlag perché in possesso delle caratteristiche necessarie ad uno studio come il nostro, ovvero disponibilità dei sorgenti e facilità nella raccolta di dati e nell'esecuzione dei test vista la tipologia multiplayer e free del gioco.

Dopo uno studio preliminare del codice dell'applicazione e del gioco sono state implementate diverse features, alcune atte a rendere l'ambiente pronto per il Machine Learning, altre atte a dimostrare la fattibilità e l'utilità di agenti intelligenti, altre ancora atte a ricavare dall'ambiente dati utili in sviluppi futuri.

Tra tutte le implementazioni, soltanto il "Controllo di Minaccia" da risultati immediatamente osservabili fornendo un aiuto intelligente al giocatore, tutte le altre vanno viste in un'ottica più ampia di questa specifica tesi. Il progetto BZFlag è lungi dall'essere portato a termine poiché le possibilità offerte dal connubio fra questo specifico gioco e l'intelligenza artificiale sono ben lontane dall'essere esaurite.

Questo lavoro è volto oltre a dare le prime dimostrazioni, a porre le basi per sviluppi futuri.

Capitolo 1

Introduzione

1.1 AI nei giochi

Il mercato mondiale di videogiochi ha fruttato nel 2004 circa 23 milioni di Euro e gli esperti prevedono che entro il 2007 si arriverà a 28 milioni sorpassando addirittura gli incassi dell'industria cinematografica.

È dunque innegabile l'importanza di questo settore dell'informatica, che si rivela essere, tra l'altro, uno dei motori dell'industria hardware oltre che software, in quanto una delle cause maggiori di “aggiornamento hardware” dei personal computer è proprio quello di permettere l'utilizzo di nuovi giochi.

Al giorno d'oggi, al contrario di fino a circa 10 anni fa, nella larghissima maggioranza dei giochi per computer a dettare la validità del videogame è la sua Intelligenza Artificiale (AI) oltre al suo aspetto grafico; l'AI, in particolare, fa sì che il gioco risulti divertente, lo trasforma in una sfida da vincere e ne detta la longevità (nonché gli incassi).

Va fatta un'importante distinzione fra l'AI “accademica” e quella “nei giochi”. La prima si divide a sua volta in *strong* e *weak*: *strong* è l'AI intesa a simulare e copiare le capacità umane, *weak* è quella che si occupa di applicare le tecnologie di AI alla soluzione di *problemi del mondo reale*.

In entrambi questi due casi si tratta di AI *ottime*, che cercano la soluzione ottima ad un problema senza troppo curarsi degli aspetti prestazionali o hardware (eseguendo per esempio calcoli per settimane su di un multiprocessore o altro).

Al contrario nelle AI applicate ai videogiochi si è costretti a fare i conti con prestazioni, tempi, ed hardware di basso livello rispetto alle possibilità accademiche, ovvero l'hardware di un Personal Computer; per questo le AI nei giochi sono sempre frutto di compromessi.

CAPITOLO 1

Nei videogiochi si parla di Intelligenza Artificiale ma spesso non ci si trova davanti ad una vera intelligenza, quanto piuttosto ad un'illusione di intelligenza; i designers del famoso videogioco Halo hanno eseguito diversi test per studiare questo aspetto. Facendo giocare alcuni giocatori ad una beta di Halo mutando fra un test e l'altro soltanto il numero di punti ferita del nemico, i risultati dei test mostrano come i giocatori ritenevano di star giocando contro AI diverse, mentre in realtà mutava soltanto il numero di punti ferita. Questo dimostra quanto possa essere facile dare ai giocatori l'illusione di diverse intelligenze artificiali, e spesso nei giochi avviene proprio così: al mutare della difficoltà mutano soltanto i punti ferita dei nemici, o le probabilità di avere tiri di dadi favorevoli, o la probabilità di uscita di alcune carte da un mazzo, piuttosto che la frequenza di imprevisti negativi o altro.

D'altra parte questi espedienti non sono sempre soddisfacenti perché per l'appunto non si tratta di agenti intelligenti e talvolta danno origine a situazioni irreali o poco divertenti.

Perché non bisogna dimenticarsi che lo scopo principale di un videogioco è il divertimento dei giocatori, e per questo il gioco non deve essere troppo difficile poiché un giocatore che perde sempre non si diverte, ma non può essere neppure troppo facile o "stupido" perché in tal caso il giocatore non vede più il gioco come una sfida alla propria abilità.

Quello che si è notato essere un aspetto critico delle AI odierne risulta essere la ripetitività di certe situazioni e la mancata capacità dell'AI di rispondere efficacemente a determinate azioni del giocatore, le debolezze dell'AI una volta individuate portano sempre il giocatore alla vittoria, la stessa mossa del giocatore anche se eseguite 100 volte produce la stessa reazione nell'AI e questo porta inevitabilmente all'appiattimento del gioco, limitandone la longevità e stancando il giocatore che non trova più sfida né divertimento nel giocare.

Nel campo dei videogiochi il termine NPC (non-player-character) viene usato per tutti gli agenti autonomi controllati dal computer; il comportamento di questi agenti è spesso ripetitivo e predicabile dal giocatore, gli NPG compiono sempre le stesse azioni, le situazioni che si vengono a creare sono sempre le medesime con un conseguente calo di interesse.

La soluzione ideale a queste carenze delle AI sta chiaramente nell'apprendimento.

Se gli NPC potessero apprendere, ogni situazione non sarebbe mai uguale alla precedente, il comportamento che ha portato una volta alla sconfitta (in ogni caso, a seconda del tipo di gioco) verrebbe evitato o modificato.

Questa branca dell'AI è detta Machine Learning ed appunto è quella che si occupa dell'apprendimento da parte degli agenti (nei giochi come nell'industria e in molti campi della vita reale).

Il Machine Learning potrebbe teoricamente portare grandi vantaggi all'industria dei videogiochi grazie alla capacità di adattamento, mutamento ed evoluzione che avrebbero gli agenti dell'AI.

I sistemi di apprendimento presentano del resto grosse problematiche da risolvere perché vi siano davvero vantaggi nel loro uso.

Un primo pensiero potrebbe essere infatti quello di addestrare gli agenti in base alle azioni del giocatore, questo sistema potrebbe però generare un'AI grezza, inefficiente e ad anche decisamente stupida, infatti apprendendo da un giocatore se ne acquisiscono sia i punti di forza ma anche le debolezze.

Per evitare ciò si potrebbe addestrare l'AI a priori, prima della vendita del prodotto, ma questo potrebbe portare ad avere agenti simili a quelli delle AI scriptate, perché non si terrebbe conto delle capacità dello specifico giocatore.

È quindi evidente che tali tecniche vanno studiate e calibrate con estrema attenzione perché possano portare effettivi benefici.

Questo e la complessità degli addestramenti delle AI ha fatto sì che fin'ora praticamente nessun gioco adotti tecniche di Machine Learning. Fanno eccezione alcune riproduzioni virtuali di "giochi di società" come gli *Scacchi*, la *Dama*, il *Risiko* e soprattutto il *Backgammon* (giocatori NPC di backgammon sono riusciti a battere ripetutamente i più grandi campioni mondiali di tale gioco).

L'unico altro caso sembra essere quello di "*Black & White*" (della Lionhead), nel quale tramite Machine Learning era possibile addestrare una Creatura (un animale) che si comportava in maniera differente a seconda di come il giocatore la trattava, e proprio questo sancì ai tempi (2000) lo strepitoso successo del gioco.

1.2 BZFlag

BZFlag [8] è un videogioco multiplayer in 3D, scritto in C++ e liberamente scaricabile dal sito del gioco stesso. Si tratta di un progetto opensource di gioco multiplayer ad alta portabilità (Irix, Linux, *BSD, Windows, Mac OS X, ecc.) e la somma di tutti questi fattori è la causa della fortuna del gioco (largamente superato il milione di download del gioco dal sito).

Il nome BZFlag sta per “Battle Zone capture Flag” ed il gioco consiste nella guida di un *tank* (carro armato) in una mappa (o mondo) popolato dai tank degli altri giocatori, alcuni amici ed altri nemici.

Le finalità di gioco possono essere molteplici, difatti esistono diverse opzioni:

- *Free for All* = combattimento libero fra team (squadre) o “tutti contro tutti”, lo scopo è distruggere i tank avversari ovviamente evitando la distruzione del proprio tank;
- *Capture the Flag* = il classico “Ruba Bandiera” in cui lo scopo è rubare la bandiera del team avversario e portarla nella propria base;
- *Rabbit Hunt* = caccia al “Coniglio” in cui tutti i giocatori né inseguono uno (il Coniglio appunto).

Il gioco viene ulteriormente *condito* dalla presenza di teletrasporti, effetti climatici o ambientali e dalle famigerate Flag (bandiere) in grado di attribuire “superpoteri” o handicap ai nostri tank (armi più potenti, invisibilità sul radar, impossibilità di sterzare a destra, impossibilità di saltare, radar fuori uso, ecc.).

Sono quindi presenti tutti i tipici ingredienti di un gioco della categoria “sparatutto” che hanno riscosso tanto successo negli ultimi anni (ad esempio la serie degli *Unreal*).

Per giocare non è necessario altro che scegliere un server dalla lista di quelli disponibili, connettersi ad esso e giocare con gli altri utenti connessi a quello specifico server, secondo le impostazioni del server (impostazioni che riguardano il numero di giocatori, le finalità, i tipi di flag e molte altre cose).

In alternativa chiunque può lanciare l'applicazione server del gioco ed impostarla come preferisce creando così il proprio mondo.

La grafica, creata grazie alle librerie OpenGL, semplice ma accattivante, la gratuità del gioco ed i requisiti di sistema praticamente nulli hanno reso BZFlag un gioco famoso negli

ambienti di lavoro o universitari dove non è possibile lo svago con giochi dagli onerosi requisiti o le grosse installazioni.

La connotazione opensource del progetto BZFlag ha reso possibile la nascita di una community di appassionati del gioco, esperti di C++, in grado di apportare grandi vantaggi ed innovazioni al gioco stesso; soprattutto in campo grafico i miglioramenti operati dalla community con l'inserimento dei tileset ed in campo amministrativo con l'aggiunta di numerose opzioni per gli admin dei server.

Diverse delle caratteristiche sopracitate hanno reso questo gioco adatto allo studio ed all'applicazione in esso di tecniche di Machine Learning per lo *studio di tattiche di gioco* e la realizzazione di *aiuti al giocatore*, ed eventualmente la realizzazione di un giocatore Computer in un ambiente interamente popolato da giocatori umani.

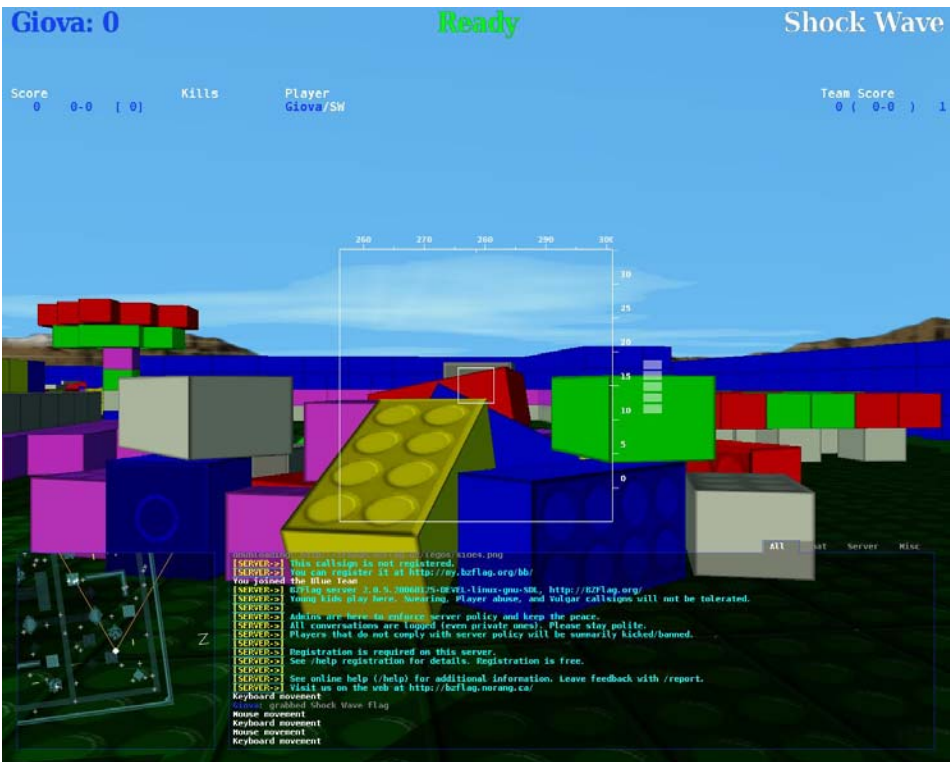


Fig. 1.1 - Mappa di BZF con texture Lego



Fig. 1.2 - Mappa di BZF con texture

Capitolo 2

Support Vector Machine

2.1 Introduzione

Le Support Vector Machine (SVM) sono una nuova e potente tecnica di classificazione sviluppata da Vapnik nella prima metà degli anni 90 nell'ambito della Statistical Learning Theory a scopi industriali.

I primi scopi per cui vennero usate furono: OCR; riconoscimento di immagini, oggetti e volti; riconoscimento vocale.

Al giorno d'oggi inoltre trovano largo utilizzo nel campo della bioinformatica, oltre che negli altri settori che prevedono l'utilizzo del Machine Learning.

Le più importanti proprietà delle SVM sono:

- difficoltà di andare in overfitting;
- possibilità di gestire dati con molte features;
- compattamento delle informazioni nel *data set* di input.

2.2 Fondamenti teorici

Una SVM classifica esempi appartenenti a classi diverse e generalizza poi su dati nuovi riuscendo a classificarli.

Le SVM possono venir usate come addestratori per classificatori Polinomiali, RBF, o Percettroni Multistrato.

Per fare questo una SVM proietta gli esempi in uno spazio (a dimensione variabile) e cerca i possibili iperpiani che separano gli esempi presenti nello spazio. Tra tutti gli iperpiani possibili viene scelto quello che massimizza la distanza dagli esempi delle classi, ed è detto *iperpiano di separazione*.

L'apprendimento consiste nel trovare appunto l'iperpiano di separazione migliore per gli esempi, che poi verrà usato anche per i casi nuovi.

Gli esempi più vicini all'iperpiano di separazione vengono detti *vettori di support* (support vector) ed è da essi che le SVM prendono nome.

Gli iperpiani sono rappresentazione spaziale delle *ipotesi*, funzioni che assegnano i dati rappresentati nello spazio R^N ad una delle classi possibili.

L'iperpiano di separazione migliore è quello che ottimizza l'*errore teorico* di quella ipotesi, dove l'errore teorico indica la bontà di quell'ipotesi nell'assegnare un esempio x (nello spazio si tratta di un punto) ad una classe y . Le ipotesi possono essere ad esempio delle RBF o dei Percettroni Multistrato.

In realtà non è possibile calcolare l'errore teorico poiché la distribuzione di probabilità del problema non è nota, ma è possibile calcolare un'approssimazione dell'errore teorico conoscendo la distribuzione dei soli esempi disponibili (*training set*), questo errore approssimato è l'*errore empirico*.

Secondo la Legge dei Grandi Numeri l'errore empirico converge a quello teorico, per cui si cercano le ipotesi che minimizzano l'errore empirico.

2.2.1 Dimensione VC

Si dice dimensione VC (Vapnik – Chervonenkis) di un classificatore (o ipotesi) f , il numero naturale che corrisponde al più grande numero di punti che possono essere separati in tutti i possibili modi dall'ipotesi f . Nel caso di due classi possibili e di k punti, i modi possibili sono 2^k .

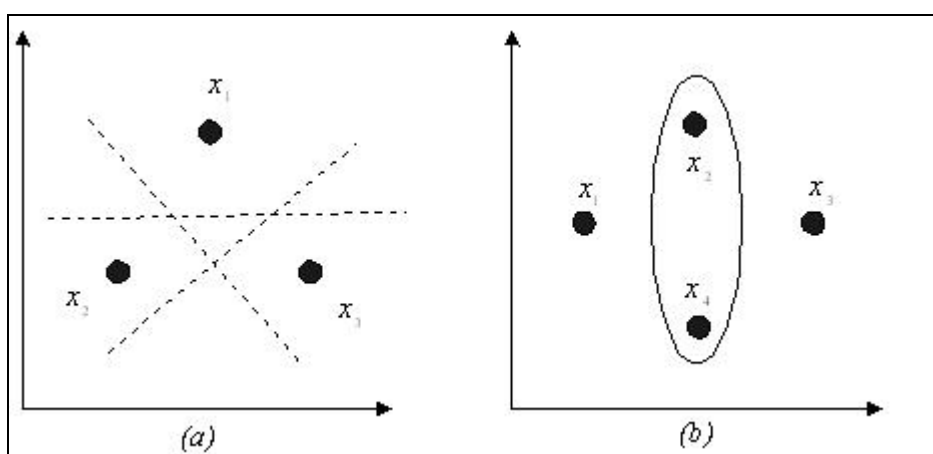


Fig. 2.1 – (a) I tre punti nel piano possono essere separati in tre modi da una retta; (b) i punti x_2 e x_4 non possono venir separati in alcun modo da x_1 e x_3 tramite una retta.

Intuitivamente la dimensione VC di un insieme di funzioni è una misura della complessità di tale insieme (della sua capacità di separazione di dati).

2.2.2 Minimizzazione dell'errore teorico

Per ottenere l'errore teorico minimo non basta minimizzare l'errore empirico ma è necessario minimizzare anche il rapporto fra la dimensione VC e il numero di punti.

La SRM (Structural Risk Minimization, in altre parole la minimizzazione dell'errore teorico) si è dimostrata essere più efficiente della classica minimizzazione del rischio empirico utilizzata nelle reti neurali.

Nelle SVM si risolve questo problema di ottimizzazione minimizzando contemporaneamente la dimensione VC e il numero di errori sul *training set*.

2.3 Classificatore lineare

2.3.1 Il caso linearmente separabile

Si tratta del caso più semplice.

Si consideri un training set e si supponga che i pattern siano separabili da un iperpiano, si vuole trovare il miglior iperpiano separatore.

Un insieme di dati si dice linearmente separabile quando è possibile trovare una coppia (w, b) tale che:

$$wx_i + b \geq +1 \quad \text{se } x_i \in \text{Classe1}$$

$$wx_i + b \leq -1 \quad \text{se } x_i \in \text{Classe2}$$

dove x_i è uno dei punti rappresentati, e $wx_i + b$ rappresenta un iperpiano.

La distanza tra il punto x_i e l'iperpiano è

$$d(x, w, b) = \frac{|xw + b|}{\|w\|}$$

e quindi la distanza tra l'iperpiano ed il punto più vicino è in funzione di $1/\|w\|$.

Quando i dati, come in questo caso, sono linearmente separabili si cerca fra tutti gli iperpiani corretti (ovvero che classificano correttamente gli esempi) quello di margine massimo rispetto ai punti e quindi di norma minima (la norma $\|w\|$ è un vettore perpendicolare).

Se la norma è piccola anche la dimensione di VC è piccola.

Quando si ha un margine ampio i rischi di overfitting sono inferiori.

Il problema di massimizzazione del margine può essere risolto grazie all'applicazione dei *moltiplicatori di Lagrange*.

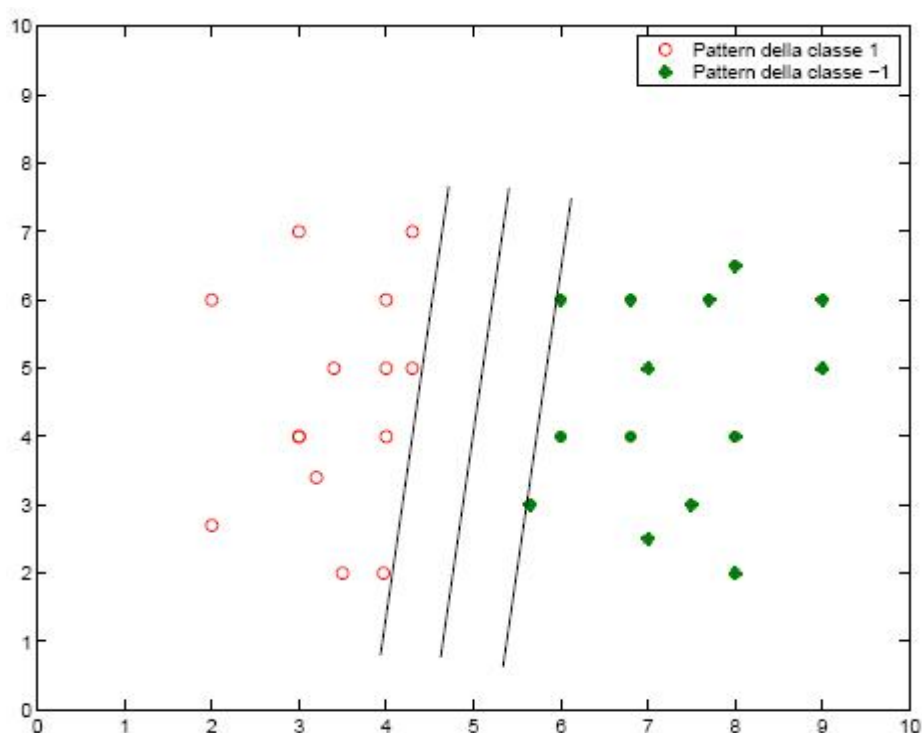


Fig. 2.2 - Iperpiano separatore per il caso linearmente separabile

2.3.2 Il caso non linearmente separabile

Si tratta di dati tra i quali alcuni sono in posizioni anomale rispetto agli altri della stessa classe. Si indica con ξ una *costante di scarto*, grande quanta è la distanza del più lontano dei punti anomali dall'iperpiano separatore.

Con l'inserimento di questa costante nei vincoli si ha una tolleranza (ξ_i per l'appunto) agli errori e perché un punto sia classificato male il suo ξ_i deve essere superiore a 1.

Si noti inoltre che $\sum_i \xi_i$ indica il limite superiore al numero di errori di training.

In questi casi in sostanza si cerca di minimizzare $\|w\|$ ed allo stesso tempo classificare i dati ma con il minor numero di errori possibile.

La soluzione al problema è la stessa che nel caso linearmente separabile eccettuato per un vincolo che limita superiormente i moltiplicatori.

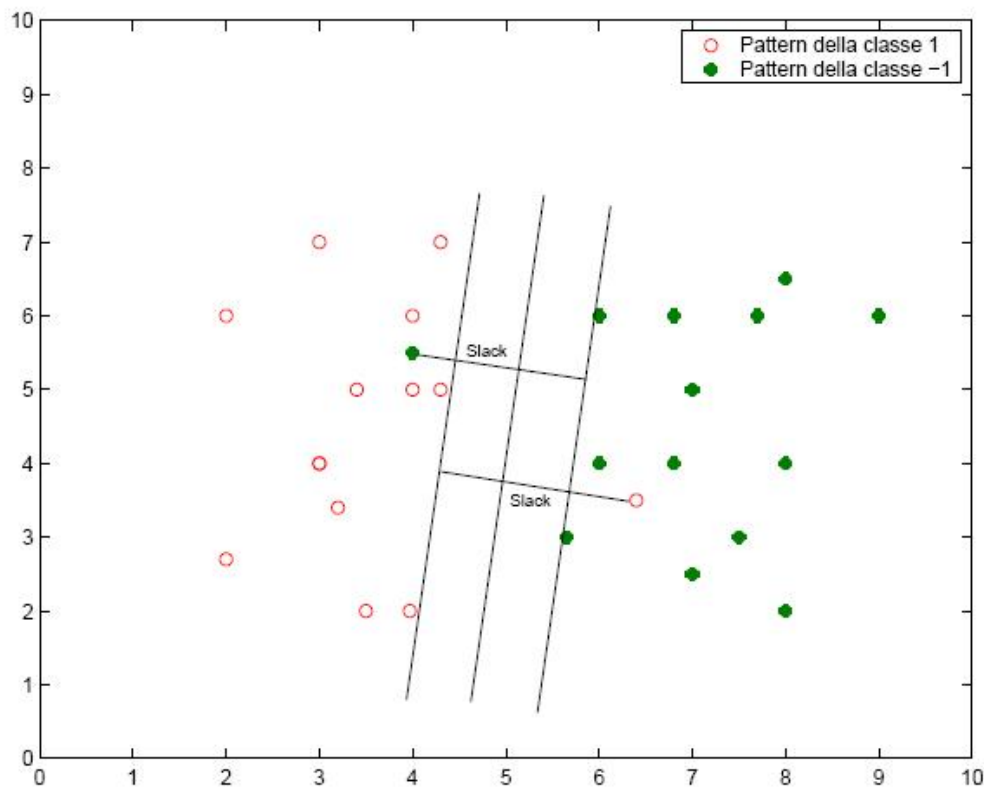


Fig. 2.3 - Iperpiano separatore per il caso non linearmente separabile

2.4 Macchine non lineari

Non sempre i problemi ricadono in una delle tipologie fin'ora presentate, in molti casi problemi non separabili nello spazio di input divengono separabili in uno spazio H (spazio di Hilbert) a maggiore dimensionalità. Il mapping da uno spazio all'altro avviene tramite una funzione $\phi: R^d \rightarrow H$. In questo modo è possibile mappare i dati del problema in uno spazio in cui siano linearmente separabili.

Avendo a che fare con uno spazio con maggior numero di dimensioni però l'algoritmo si trova a lavorare con vettori di grandi dimensioni generando problemi dal punto di vista dei tempi e dell'efficienza di calcolo. Si può ovviare a tutto ciò utilizzando una funzione di

Kernel, in questo modo l'algoritmo lavora a tutti gli effetti nello spazio H ed ha gli stessi tempi e performance che nello spazio di input.

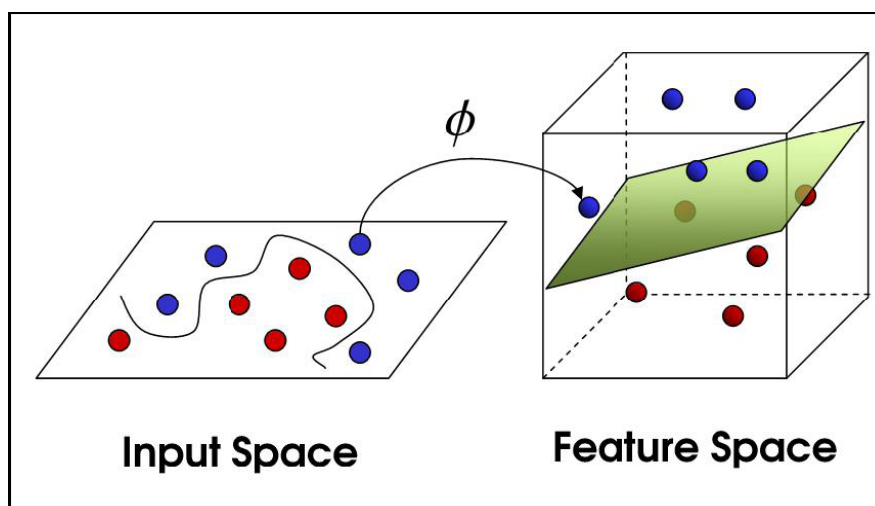


Fig. 2.4 - Mapping in uno spazio a dimensione superiore

È necessario scegliere con oculatezza il tipo di Kernel utilizzato poiché l'uso di Kernel non appropriati al problema potrebbe portare all'overfitting.

A questo punto si ha a che fare con un problema separabile linearmente (in H ovviamente) e che quindi ricade in una delle casistiche già illustrate.

Classificatore	Funzione Kernel
Lineare	$K(\vec{x}, \vec{y}) = \vec{x} \cdot \vec{y}$
Polinomiale di grado d	$K(\vec{x}, \vec{y}) = (\vec{x} \cdot \vec{y} + 1)^d$
Gaussiano	$K(\vec{x}, \vec{y}) = \exp(-1/2\sigma^2 \ \vec{x} - \vec{y}\ ^2)$
Percettrone Multistrato	$K(\vec{x}, \vec{y}) = \tanh(\vec{x} \cdot \vec{y} - \theta)$

Tab. 2.1 - Funzioni Kernel

Capitolo 3

LIBSVM

3.1 Caratteristiche generali

LIBSVM [12] consiste in un tool per la classificazione (C-SVC, nu-SVC), regressione (epsilon-SVR, nu-SVR) e stima della distribuzione (SVM ad una classe).

Un obiettivo importante di questa libreria è quello di permettere agli utenti, non solo informatici, un facile utilizzo delle SVM.

Il package è costituito da tre tool a linea di comando: *svm-train*, *svm-predict* ed *svm-scale* e dalla libreria *LIBSVM* contenente la classe *svm*.

Il primo modulo è utilizzato per addestrare l'SVM attraverso un training set, il secondo per testare l'SVM attraverso un test-set, il terzo modulo è utilizzato per effettuare una normalizzazione dei dati nel range definito dall'utente, che può essere ad esempio $[1, -1]$ oppure $[0, 1]$.

Il tool è in grado di risolvere efficacemente cinque problemi:

- C-Support Vector Classification
- ν -Support Vector Classification
- ϵ -Support Vector Regression
- ν -Support Vector Regression
- Distribution Estimation (one-class SVM)

```

public class svm
{
    public static svm_model svm_train(svm_problem prob,
        svm_parameter param);
    public static void svm_cross_validation(svm_problem prob,
        svm_parameter param, int nr_fold, double[] target);
    public static int svm_get_svm_type(svm_model model);
    public static int svm_get_nr_class(svm_model model);
    public static void svm_get_labels(svm_model model, int[] label);
    public static double svm_get_svr_probability(svm_model model);
    public static void svm_predict_values(svm_model model,
        svm_node[] x, double[] dec_values);
    public static double svm_predict(svm_model model, svm_node[] x);
    public static double svm_predict_probability(svm_model model,
        svm_node[] x, double[] prob_estimates);
    public static void svm_save_model(String model_file_name,
        svm_model model) throws IOException
    public static svm_model svm_load_model(String model_file_name)
        throws IOException
    public static String svm_check_parameter(svm_problem prob,
        svm_parameter param);
    public static int svm_check_probability_model(svm_model model);
}

```

Tab. 3.1 – Classe *svm* della LIBSVM

I passi fondamentali nell'utilizzo del tool sono i seguenti:

1. Convertire i dati in input nel formato del tool che si intende utilizzare.
2. Effettuare la normalizzazione dei dati nel range opportuno
3. Scegliere il tipo di funzione kernel da utilizzare
4. Utilizzare la cross-validation per determinare i parametri *soft-margin* e *ampiezza della gaussiana*
5. Con i valori trovati con la cross-validation, rieffettuare la fase di training

3.2 Formato dei dati

Il formato utilizzato per i campioni di addestramento e di test è visibile nella Tabella 3-1.

Ogni linea rappresenta un campione ed i commenti sono preceduti dal carattere cancelletto.

Ogni campione è costituito dal valore della funzione target per l'esempio considerato e da una serie di coppie *feature* : *valore*.

Il valore della funzione target per l'esempio considerato può assumere i valori +1, -1, 0 o un qualsiasi valore floating point .

Le features sono rappresentate da valori sia interi sia di tipo qid, ovvero un valore speciale usato nella modalità ranking, mentre i loro valori sono di tipo floating point.

Il valore target ed ogni coppia feature-valore sono separati da uno spazio bianco.

Le coppie feature-valore sono disposte in ordine crescente per valore di feature e le features con valore 0 possono essere saltate.

Un campione con valore target pari a 0, indica che l'esempio deve essere classificato utilizzando la transduzione.

```

<linea> = <target> < featurei : valorei> ... < featuren : valoren> #info
<target> = +1 | -1 | 0 | <float>
<feature> = <int> | quid
<valore> = <float>
<info> = <string>

```

Tab. 3.2 – Formato dei dati per la LIBSVM

3.3 Normalizzazione

Supponendo di aver eseguito precedentemente la fase di preprocessing, quindi di avere il data set nel formato adottato dal tool, è possibile passare alla fase di normalizzazione

dei dati tramite il comando `svm-scale`. L'obiettivo di questa fase è quello di effettuare dei piccoli aggiustamenti ai dati allo scopo di renderli più adatti alle valutazioni del kernel.

Supponendo ad esempio di prendere in considerazione un Kernel Gaussiano - un particolare tipo di kernel che fa parte di quelle funzioni kernel chiamate RBF (Radial Basis Functions). Se una certa feature di un campione, presenta un maggiore range di variazione rispetto ad un'altra, allora questa dominerà la sommatoria nella gaussiana, mentre feature con piccole variazioni di

range saranno essenzialmente ignorate. Da questo se ne può ricavare che feature con maggiore range di variazione, avranno maggiore attenzione da parte dell'algoritmo SVM. È necessaria quindi una normalizzazione dei valori delle feature in modo tale che questi cadano all'interno dello stesso range, che può essere ad esempio $[0,1]$ oppure $[-1,+1]$.

Questa normalizzazione può essere eseguita attraverso il comando `svm-scale` che accetta in input un data set e restituisce come output lo stesso data set normalizzato nel range scelto e un insieme di parametri di scalatura. Questi dati riscalati verranno poi utilizzati dal modulo che realizza l'apprendimento.

Il formato del comando di scalatura è il seguente:

`svm-scale -s param scalatura training file > training file scalato`

Per specificare il range di scala, vengono utilizzate due opzioni che specificano rispettivamente il lower bound e l'upper bound del range, ovvero `-l` e `-u`.

3.4 Addestramento e Test

Come si può notare la cross validation non utilizza tutti gli esempi di training, infatti ad ogni iterazione alcuni di questi vengono esclusi dalla valutazione. È necessario quindi effettuare l'addestramento sull'intero training set utilizzando i parametri che sono stati determinati dalla precedente fase.

La fase di training avviene tramite il comando `svm-train` nel seguente modo:

```
svm - train [-options] training file scalato model file
```

Il tool fornisce diverse opzioni di training (per una descrizione più accurata si rimanda alla documentazione ufficiale).

Per prima cosa il tool permette di scegliere quale tipo di problema risolvere, quindi quale tipo di SVM utilizzare, attraverso l'opzione "-s".

È possibile scegliere tra:

- C-SVC
- ν-SVC
- one-class SVM
- γ -SVM
- ν-SVR

Tramite l'opzione "-t" è possibile selezionare uno dei kernel seguenti:

- Lineare
- Polinomiale
- Radial Basis Function (kernel di default)
- Sigmoidale

Una volta terminata la fase di training, si ha a disposizione la funzione decisionale appresa memorizzata nel model file e i dati di test a loro volta scalati. A questo punto è possibile effettuare le predizioni delle label per gli esempi di test attraverso il comando `svm-predict` che ha la seguente forma:

```
svm - predict [-options] test file scalato model file predictions
```

Come è possibile notare, l'input del comando consiste nel test set scalato, nel model file ed il file in cui memorizzare l'output. L'output consiste nelle predizioni delle label, $\text{sgn}(f(x))$, del test-set.

Capitolo 4

Interfaccia

4.1 Introduzione

Implementando la classe *NetworkInterface* sono state introdotte diverse funzionalità nel gioco, tutte legate al Machine Learning, che vengono attivate e disattivate tramite la pressione di alcune key preimpostate.

BZFlag interpreta come un “evento” la pressione di qualsiasi key e per tanto il controllo passa alla funzione *doEvent* atta a gestire tutti i tipi di eventi. Tramite uno *switch* avviene il riconoscimento dell’evento in esame e l’evento legato alla pressione delle key è *KeyDown* della classe *BzfEvent*.

Di base questo evento veniva gestito tramite una chiamata alla funzione *doKey* che si preoccupa di eseguire l’azione legata alla key.

Per permettere la gestione delle opzioni questo *case* dello *switch* è stato profondamente modificato con l’aggiunta di uno *switch* annidato che gestisce tutte le opzioni. Ogni opzione è attivata da una specifica key individuata tramite una costante della classe *NetworkInterface* ed alla pressione della key vengono eseguiti i metodi della classe corrispondenti.

Nel caso in cui la key non sia legata ad una delle opzioni, allora viene passata alla funzione *doKey*.

```

case BzfEvent::KeyDown:
    if(netInt.GetOptionsFlag() == false) {
        if(event.keyDown.ascii == KEY_OPTIONS)
            netInt.ChangeOptionsFlag();
        else doKey(event.keyDown, true);
    }
    else {
        switch(event.keyDown.ascii) {
            case KEY_OPTIONS:
                netInt.ChangeOptionsFlag();
                break;
            case KEY_ENEMYMAP:
                netInt.ChangeEnemyFlag();
                break;
            case KEY_CONTROL:
                netInt.ControlEnemy(curMaxPlayers, player);
                break;
            case KEY_INIT_READ:
                netInt.InitInputFile();
                break;
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                netInt.RunThread(event.keyUp.ascii);
                break;
            case KEY_DUMP:
                netInt.SaveDataTank(curMaxPlayers, player, );

                break;
            case KEY_SAVE:
                netInt.ChangeSaveFlag();
                break;
            case KEY_TRAIN:
                if(netInt.GetTrainFlag() == false) {
                    netInt.ChangeTrainFlag();
                    netInt.TrainThread(1);
                }
                else netInt.ChangeTrainFlag();
                break;
            case KEY_TEST:
                netInt.TrainThread(3);
                break;
            default:
                doKey(event.keyDown, true);
                break;
        }
    }
    break;

```

Tab. 4.1 – Funzione *doEvent*: gestione delle chiamate alle opzioni aggiunte.

Ecco l'elenco delle opzioni aggiunte:

- attiva Opzioni
- salva Dati Tank
- salva Azioni
- leggi Azioni da file
- salva Dati Train
- controllo Nemici manuale
- controllo Nemici automatico
- salvataggio Mappa

4.2 Attivazione Opzioni

Quasi la totalità delle opzioni aggiunte al gioco BZFlag viene attivata tramite la pressione di una key da tastiera. Questo fatto va in conflitto con la possibilità di chattare durante il gioco con gli altri giocatori, infatti diverse key invece di aggiungere una lettera al messaggio di chat da inviare, attivano una delle opzioni aggiunte.

Per porre rimedio a questo problema è stata aggiunta una nuova funzionalità, “l’attivazione Opzioni” per l’appunto.

Questa funzionalità fa sì che le key siano legate alle opzioni e non hai normali caratteri solo se prima è stata premuta la KEY_OPTIONS.

Una pressione di tale key attiva le opzioni, mentre un’ulteriore pressione le disattiva riattivando i caratteri.

Ovviamente la KEY_OPTIONS andrà legata al tasto di un carattere poco usato altrimenti si avrà lo stesso problema per cui questa funzionalità è stata implementata.

Nella funzione *doEvent* prima di eseguire lo *switch* sulle key delle opzioni, viene fatto un check sulla *optionsFlag* della classe *NetworkInterface*, flag il cui stato è legato alle pressioni della KEY_OPTIONS.

Se tale flag è attiva allora si esegue il controllo sulle key delle opzioni, altrimenti le key vengono trattate come normali caratteri dalla funzione *doKey*.

(default: KEY_OPTIONS = “+”)

4.3 Salvataggio Dati Tank

Tramite la pressione della KEY_DUMP viene eseguito il metodo *SaveDataTank* della classe *NetwokInterface*, questo metodo ha diverse funzionalità a seconda dei parametri che riceve in input.

Tra queste funzionalità vi è quella di raccogliere e calcolare i dati di ogni tank presente in gioco (compreso *MyTank*) di memorizzarli nell'apposita struttura *dataTank* e di salvarli sul file indicato dalla costante BZ_DATA.

La scrittura sul file avviene in modalità *append* e quindi è possibile salvare i dati dei tank in diversi momenti, contraddistinti da data ed ora.

(default: KEY_DUMP = "p")

(default: BZ_DATA = "bzflag_data.txt")

4.4 Salvataggio Azioni

Premendo la KEY_SAVE si cambia lo stato della *saveFlag*; quando questa è settata su TRUE ogni key processata dalla funzione *doKey* viene memorizzata nell'apposita struttura *azione* dal metodo *NetworkInterface::SaveAction*.

Questo memorizza nella struttura le informazioni utili legate a quella key ed aggiunge ogni azione ad una coda di azioni.

Quando viene premuto il tasto "Esc" tutta la coda viene scritta in modalità *append* sul file BZ_OUTPUT dal metodo *OutputActionsFile*.

Nel caso in cui la flag sia settata a FALSE invece non vi è alcuna memorizzazione.

(default: KEY_SAVE = "s")

(default: BZ_OUTPUT = "bzflag_out.txt")

4.5 Lettura ed esecuzione Azioni

Una delle funzionalità aggiunte consiste nella possibilità di far eseguire al tank azioni preimpostate e scritte in appositi file.

Lo scopo è rendere possibile la scelta di una serie di azioni (o di una strategia) fra diverse disponibili, una per ogni file (massimo 10).

Il tasto KEY_INIT_READ causa l'esecuzione del metodo *NetworkInterface::InitInputFile*. Questo metodo esegue il metodo *InputFile* per ogni file presente nella cartella dell'applicazione e con il nome rispondente ai requisiti.

BZ_INPUT + num + ".txt"
 num >= 0 ; num < 10
 esempio: "bzflag_in7.txt"

Tab. 4.2 – Forma dei nomi dei file di input.

Quando non vengono più trovati file di input idonei vengono allocati un vettore di thread ed uno di *long* con tanti elementi quanti erano i file di input trovati, questi vettori serviranno per l'esecuzione delle azioni.

Il metodo *InputFile* oltre ad assemblare il nome del file da leggere ed a verificarne l'esistenza, se il file è presente procede con la lettura di quanto vi è scritto.

Il file viene scorso dall'inizio fino al carattere EOF (End Of File) ed ogni riga letta viene memorizzata in un'istanza della struttura *azione*, infine questo elemento viene inserito in una coda identificata dallo stesso numero che identificava il file di origine.

A questo punto, dopo aver predisposto una queue per ogni file di input, il sistema è pronto a replicare le azioni memorizzate.

Ciò avviene chiamando una delle code tramite la pressione di una delle key numeriche (da 0 a 9), infatti così facendo dalla funzione *doEvent* viene invocato il metodo *RunThread*. Tale metodo memorizza in un elemento del vettore di *long* precedentemente allocato il tempo attuale a partire da cui verranno calcolati i tempi delle singole azioni, e lancia un

CAPITOLO 4

thread associandolo alla coda tramite il suo numero identificativo. Il thread esegue la funzione *PlayActions* che è responsabile dell'esecuzione delle azioni delle code.

La coda associata al thread che esegue la funzione viene passata elemento per elemento, ogni elemento corrisponde ad un'azione e contiene le informazioni necessarie alla sua riproduzione. Grazie alla funzione *Sleep* viene atteso il tempo memorizzato nell'azione (tempo calcolato a partire dal tempo iniziale della coda) e poi viene simulato un evento corrispondente all'azione in oggetto. L'applicazione a questo punto tratta l'evento simulato come un evento generato realmente da tastiera ed esegue l'azione.

Quando tutta la cosa è vuota il thread viene distrutto.

Mentre vengono eseguite le azioni lette da file è anche possibile eseguire azioni da tastiera e quindi si potrebbe pensare di creare liste di azioni con strategie di movimento e lasciare al giocatore la responsabilità di fare fuoco o di intervenire in caso di pericolo, e come queste possibilità ve ne sono molte altre.

Tramite la pressione dei tasti numerici è anche possibile intervallare una coda all'altra e quindi unire diverse strategie a seconda della necessità (ad esempio coda con strategia di “fuga” e coda con strategia di “attacco”).

(default: KEY_INIT_READ = “r”)

4.6 Salvataggio Dati Train

Forse l'opzione più importante tra quelle aggiunte è quella che permette tramite l'utilizzo delle SVM di classificare i tank avversari in base alla minaccia che possono costituire per il giocatore.

Per fare questo è necessario però generare un modello della soluzione tramite le SVM, modello creabile soltanto tramite la fase di addestramento.

Per eseguire l'addestramento sono necessari i dati su cui addestrare l'agente, questi dati possono venir raccolti tramite alcuni metodi implementati e che vengono attivati dalla pressione della KEY_TRAIN.

Questa key attiva la flag *trainFlag* e invoca il metodo *TrainThread* che crea un thread che esegue la funzione *GetTrainingData*.

Questa funzione utilizza la *Sleep* per invocare ad intervalli regolari (default: 10sec) il metodo *SaveDataTank* con parametro “param = 1”.

Questo fa sì che una volta che questo metodo abbia calcolato i dati *distance* e *diffTracking* li scriva con modalità *append* nel file BZ_TRAIN.

SaveDataTank continua a venir chiamato ad intervalli regolari fino a che un’ulteriore pressione della KEY_TRAIN non disattiva la *trainFlag*, al che si interrompe il salvataggio dei dati per l’addestramento.

Questi dati memorizzati su file possono poi essere utilizzati appunto per l’addestramento di un agente che sappia classificare i nemici in “pericolosi” e “non pericolosi”.

(default: KEY_TRAIN = “t”)

(default: BZ_TRAIN = “bzflag_training.txt”)

```
struct svm_node
{
    int index;
    double value;
};
```

Tab. 4.3 – Struttura dei dati usati dalla LIBSVM

4.7 Controllo Nemici Manuale

Grazie alla raccolta dati ed all’addestramento eseguito con le SVM è possibile generare un modello per la soluzione del problema della classificazione degli avversari.

È possibile utilizzare questo modello per riconoscere i nemici probabilmente pericolosi da quelli meno probabili.

Premendo la KEY_CONTROL viene effettuato un controllo istantaneo ed i probabili nemici pericolosi sono segnalati tramite un messaggio scritto nella finestra di log del gioco.

CAPITOLO 4

Viene eseguito il metodo *ControlEnemy* della classe *NetworkInterface* che calcola per tutti i tank presenti i valori di “distanza” e “differenza di puntamento” e li passa in input al metodo *CeckEnemy*.

```
bool NetworkInterface::CeckEnemy(int distance, int trackingDiff)
{
    const struct svm_model *modello = svm_load_model("BZ_MODEL");
    struct svm_node nodo[3];
    nodo[0].index = 1;
    nodo[0].value = distance;
    nodo[1].index = 2;
    nodo[1].value = trackingDiff;
    nodo[2].index = -1;
    nodo[2].value = 0;
    if(svm_predict(modello, nodo) == 1)
        return true;
    else
        return false;
}
```

Tab. 4.4 – Metodo *CeckEnemy*

Tale metodo carica il modello *BZ_MODEL* nella struttura *svm_model* della LIBSVM e memorizza i dati precedentemente calcolati in un vettore di istanze della struttura *svm_node* (sempre della LIBSVM).

Il numero di elementi di tale vettore è uguale a quello delle caratteristiche (i dati) più uno, perché è necessario un ultimo elemento con indice “-1” che segnali il termine delle caratteristiche; nel nostro caso le caratteristiche sono due (*distance* e *diffTracking*) quindi il vettore viene allocato per tre elementi.

Infine viene invocata la *svm_predict* di LIBSVM con come parametri il modello ed il vettore delle caratteristiche; la funzione restituisce “1” se il nemico è catalogato come “pericoloso”, altrimenti esso è “non pericoloso”.

Per ogni tank, se il ceck è positivo viene visualizzato nel log il nome del tank, altrimenti viene visualizzata la stringa “*****”.

(default: KEY_CONTROL = “c”)

(default: BZ_MODEL = “bzflag_training.model”)


```

RADEON 9200 Series DDR x86/MMX/3DNow!/SSE2
Message of the day:
* BZFlag is a free multiplayer multiplatform 3D tank battle game. The name stands for Batt
BSD, Windows, Mac OS X and other platforms.
You joined the Blue Team
[SERVER->] BZFlag server 2.0.4.20050930-STABLE-W32VC71, http://BZFlag.org/
Keyboard movement
[SERVER->] You are unable to begin playing for 4.0 seconds.
[SERVER->] BZFlag server 2.0.4.20050930-STABLE-W32VC71, http://BZFlag.org/
[SERVER->] You are unable to begin playing for 2.9 seconds.
Glova: blew myself up
Keyboard movement
Controllo nemici:
*****
Controllo nemici:
Mysa
Controllo nemici:
*****
Controllo nemici:
*****

```

Fig. 4.1 – Esempio di quattro esecuzioni del Controllo Nemici Manuale: soltanto una volta viene individuato come pericoloso il giocatore “Mysa”.

4.8 Controllo Nemici Automatico

Poco sopra è stato presentato il controllo manuale della pericolosità degli avversari, ma esso non è l’unico tipo di controllo implementato, vi è anche il controllo automatico.

Tramite la KEY_ENEMYMAP si modifica lo stato della flag *enemyFlag*, tale variabile è dichiarata *static* e viene utilizzata nel metodo *static NetworkInterface::IsEnemy*.

Tale metodo viene chiamato staticamente dal metodo *render* della classe *RadarRender* che è quello responsabile del disegno del radar sullo schermo.

Il metodo *render* viene chiamato ad ogni repaint dello schermo ed esamina ogni tank in gioco per leggerne il colore con cui disegnarlo, è stato aggiunto un blocco *if* la cui condizione è proprio una chiamata a *IsEnemy*: se l’output è uguale a TRUE allora significa che il tank in questione è pericoloso e viene disegnato sul radar non con il suo vero colore ma con uno impostato per segnalare lo stato di minaccia (default: giallo ocra).

```

if(NetworkInterface::IsEnemy(player) == true)
{
    float dimmedcolor[3];
    dimmedcolor[0] = 1.0;
    dimmedcolor[1] = 0.7;
    dimmedcolor[2] = 0.0;
    glColor3fv(dimmedcolor);
}

```

Tab. 4.5 – Funzione *render*: se il tank è catalogato come pericoloso viene disegnato sul radar con un colore particolare.

Questo fa sì che ad ogni repaint i tank siano valutati, e quelli scelti come “minacciosi” siano disegnati sul radar da un cambio di colore, dal loro colore originale a quello “di minaccia”.

Il metodo *IsEnemy* una volta invocato esegue un check su *enemyFlag* e se questa è disattivata ritorna FALSE immediatamente, in questo modo i tank vengono disegnati con i loro colori di team visto che la flag, e quindi l’opzione di riconoscimento minaccia, non è attiva. Se al contrario la flag è attivata si calcolano, similmente al metodo *ControlEnemy*, i valori di distanza e differenza di puntamento per il tank ricevuto in input e si passano questi parametri al metodo *CeckEnemy*. Sarà poi questo metodo, come spiegato in precedenza, a valutare la minaccia del tank in oggetto, il risultato di questa valutazione sarà ritornato al metodo *render* che disegnerà in base ad esso il tank.

(default: KEY_ENEMYMAP = “e”)

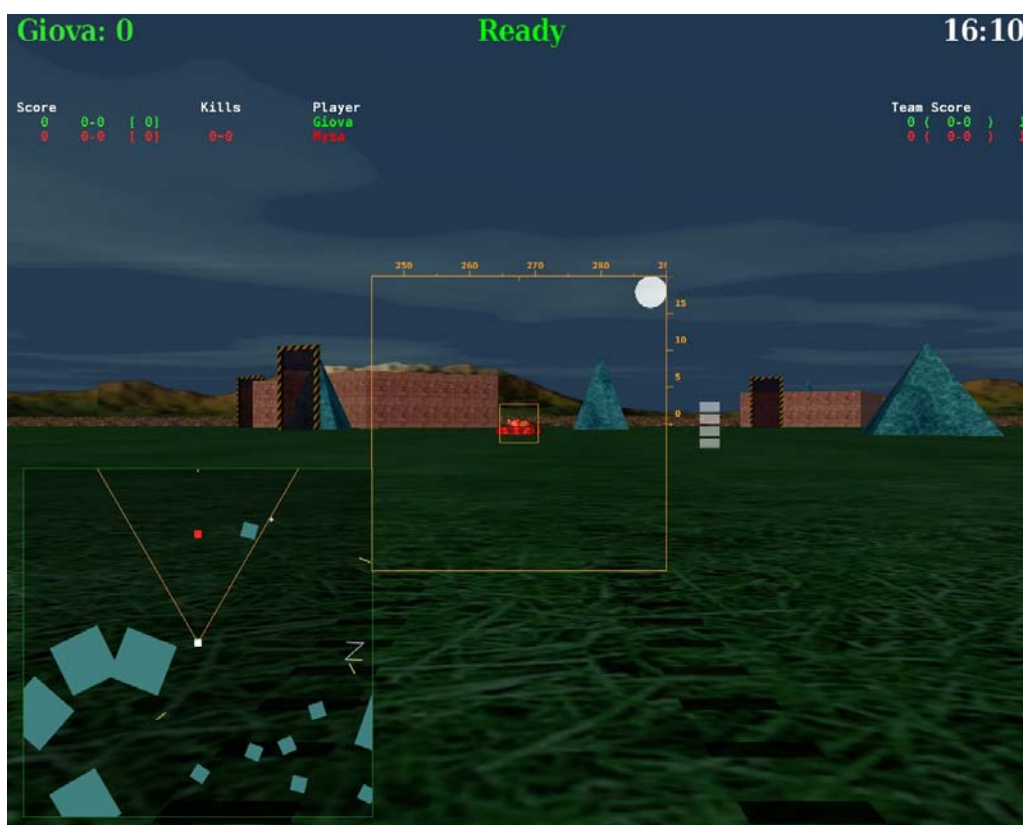


Fig. 4.2 - Controllo Nemici Automatico: il tank è lontano ed è nel colore del suo team (rosso)

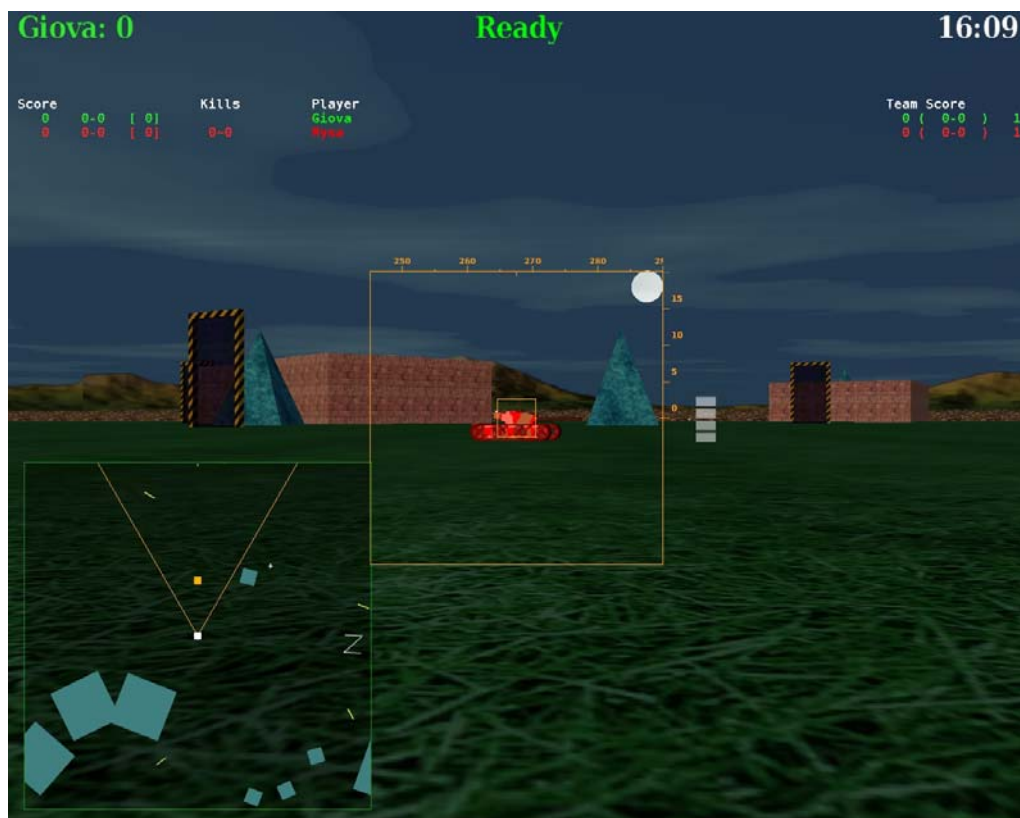


Fig. 4.3 - Controllo Nemici Automatico: il tank è ora più vicino ed è nel colore "minaccia" (giallo)

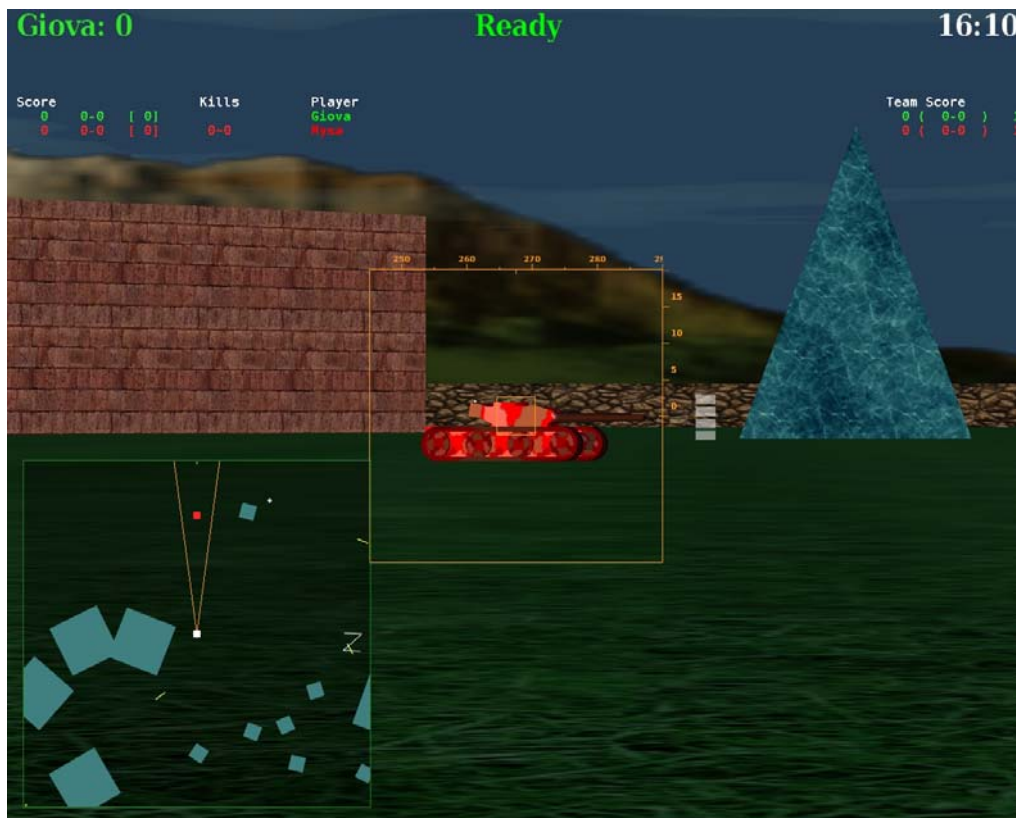


Fig. 4.4 - Controllo Nemici Automatico: il tank punta lontano da noi (rosso)

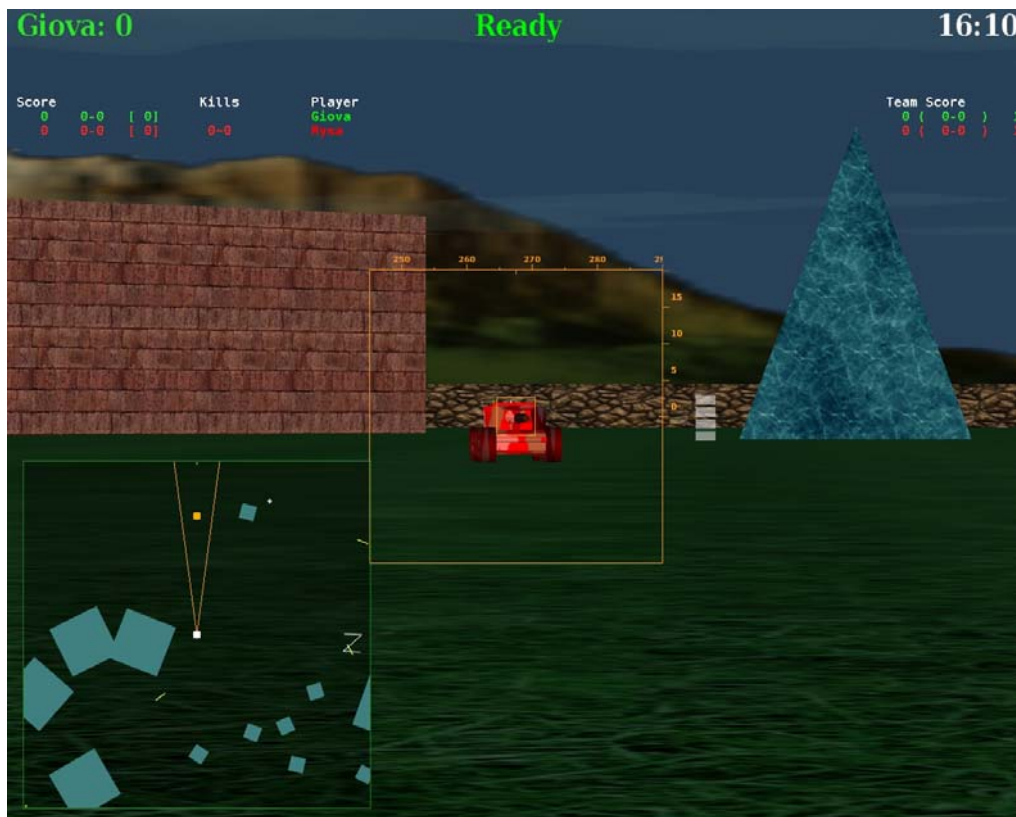


Fig. 4.5 - Controllo Nemici Automatico: il tank punta vicino a noi quindi è una “minaccia” (giallo)

4.9 Salvataggio Mappa

Unica fra tutte le opzioni aggiunte ad essere eseguita in automatico, senza bisogno di un comando del giocatore, è il salvataggio della mappa.

La mappa è il mondo di gioco, uno spazio 3D (un parallelepipedo) in cui i tank si muovono, combattono e giocano fra ostacoli, portali, flag ed altro.

BZFlag fa sì che ogni tank sia dotato di un radar raffigurante tutto ciò in uno spazio 2D, in modo che il giocatore possa con una rapida occhiata conoscere la situazione attuale di gioco ed elaborare le proprie strategie.

Il disegno del radar nella finestra di gioco avviene tramite la classe *RadarRender*, nella quale è stato inserito l'uso della classe *NetworkMapper* da noi realizzata.

```
class NetworkMapper
{
private:
    int **map;
    int maxX, maxY;
    bool mapperFlag;
    int xDim, yDim;
    int num;
    int count;

    void Paint();
    void PaintFill(int punti[4][2]);
    void FloodFill(int x, int y, int **bufferMatrix);
    void Bresenham(int p0[2], int p1[2], int** m);
    void WritePixel(int x, int y, int** m);
    void OutputMapFile(int **mappa, int mX, int mY, bool error);
    void Init(int width, int height);
    bool CeckPoint(int x, int y);

public:
    NetworkMapper();
    void Mapper();
};
```

Tab. 4.6 – Classe *NetworkMapper*.

Quando questa procedura è stata effettuata per ogni oggetto, si ha una matrice raffigurante la mappa e la si scrive sul file BZ_MAP.

La procedura di Flood-Fill è compresa in un blocco *try* per gestire eventuali eccezioni sollevate; in caso di eccezioni il blocco di mappa composto fino a quel momento viene scritto sul file BZ_MAP_ERR e si continua l'esecuzione passando direttamente all'oggetto successivo, ignorando l'oggetto che ha causato l'eccezione.

(default: BZ_MAP = "bzflag_map.txt")

(default: BZ_MAP_ERR = "bzflag_mapError.txt")

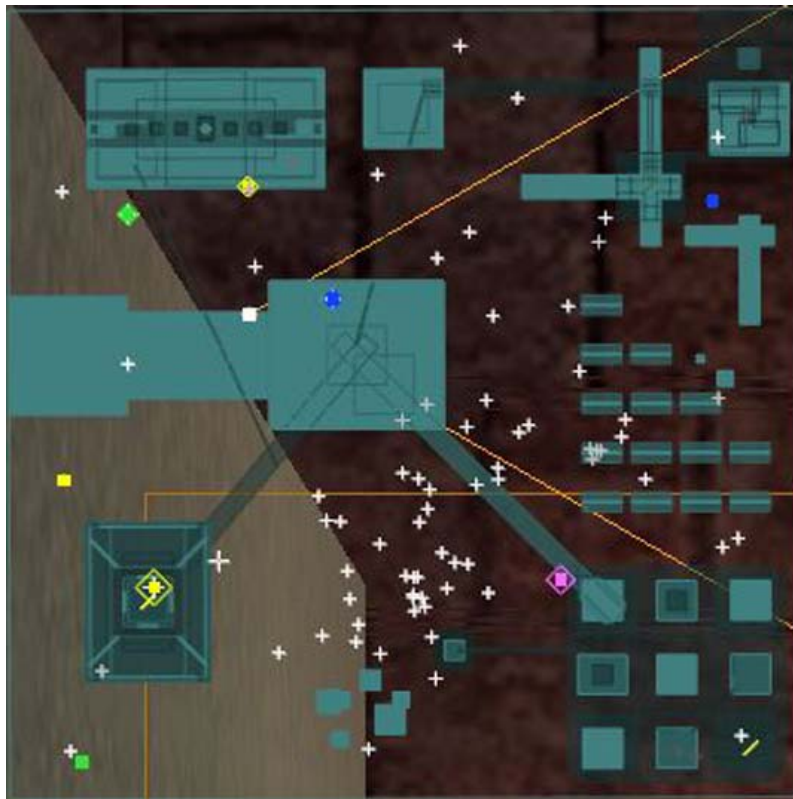


Fig. 4.7 - Esempio di mappa vista sul radar.

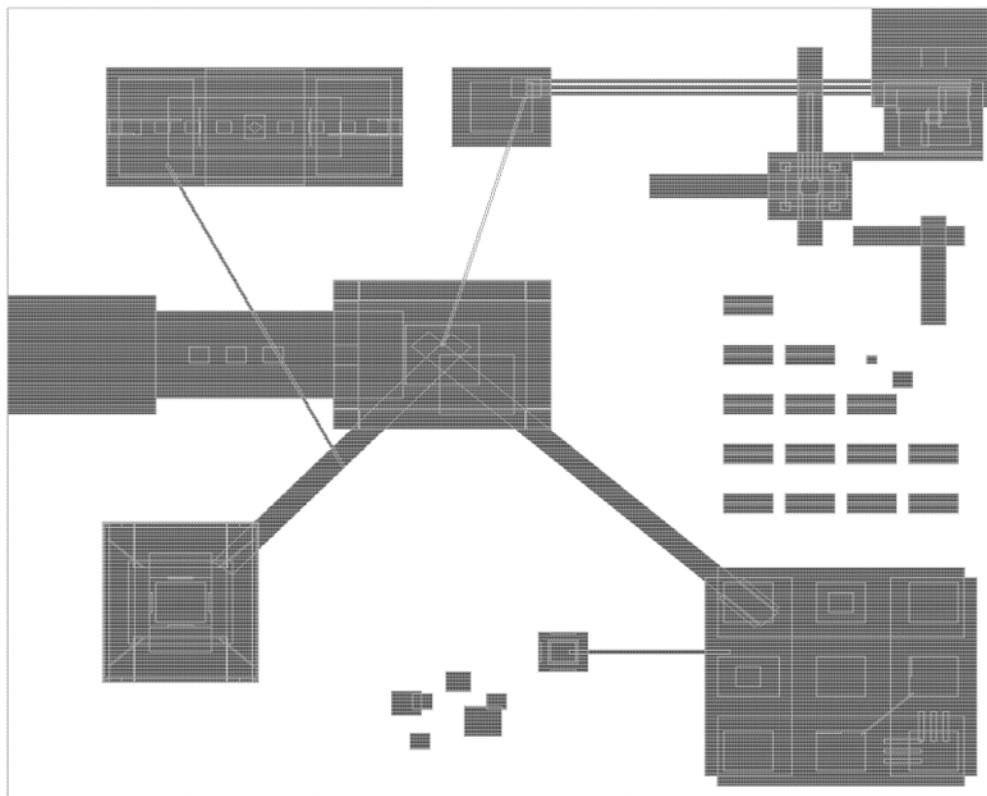


Fig. 4.8 - Rappresentazione (rimpicciolita) su file della mappa in Fig. 4.7.


```

class NetworkInterface
{
private:
    int prevAction;
    bool prevActionPress;
    long initialTime;
    bool saveFlag;
    queue<azione> actions;
    static dataTank dataMe;
    dataTank* dataPlayers;
    int numPlayers;
    pthread_t *pThread;
    queue<azione> coda[100];
    long* runInitTime;
    pthread_t trainThread;
    bool optionsFlag;
    static bool enemyFlag;
    int errorTest;
    bool trainFlag;

    void OutputDataFile(int realNumPlayers);
    void OutputTrainingFile(int realNumPlayers);
    void OutputActionsFile();
    queue<azione> InputFile(int num);
    static bool CeckEnemy(int distance, int trackingDiff);

public:
    NetworkInterface();
    void ChangeSaveFlag();
    void SaveAction(const BzfKeyEvent& key,
        bool pressed);
    static void SaveMyDataTank();
    void SaveDataTank(int _numPlayers,
        RemotePlayer** player, int param);
    void InitInputFile();
    void RunThread(int num);
    void CallThread(int num);
    queue<azione> GetCoda(int num);
    long GetRunInitTime(int num);
    bool GetOptionsFlag();
    void ChangeOptionsFlag();
    void ControlEnemy(int _numPlayers,
        RemotePlayer** player);
    void ChangeEnemyFlag();
    static bool IsEnemy(const RemotePlayer* player);
    void TrainThread(int param);
    bool GetTrainFlag();
    void ChangeTrainFlag();
    void OutputTest();
};

```

Tab. 4.7 – Classe *NetworkInterface*.

Capitolo 5

Machine Learning in BZFlag

5.1 Thread

Un thread è l'unità base nell'uso delle CPU e consiste di un program counter, un insieme di registri e uno spazio di stack.

Una programmazione multithreading permette l'esecuzione contemporanea di più parti di codice; si tratta di un'illusione poiché la CPU può eseguire una sola istruzione per volta, ma è ugualmente un metodo molto efficace.

Nel corso dello sviluppo del progetto è risultato diverse volte evidente come l'utilizzo di un sistema multithreading avrebbe permesso di semplificare notevolmente alcuni algoritmi e di ottimizzare le prestazioni di alcuni passaggi.

Per questo è stata aggiunta al progetto la libreria “*pthread*” [13] che ha permesso l'aggiunta e l'utilizzo dei thread.

Grazie ai thread è stato possibile in diversi casi simulare un timer ed eseguire metodi ad intervalli regolari, continuando fra un intervallo e l'altro la normale esecuzione dell'applicazione, senza dover bloccare tutta l'applicazione in attesa.

La funzione *pthread_create* permette di creare appunto un thread e di fargli eseguire una funzione passandole anche dei parametri, così con la scrittura di poche funzioni grazie ai parametri si è riusciti a coprire diversi casi d'uso.

5.2 Mappa

Caratteristica peculiare di ogni server di gioco è la mappa, ovvero il mondo in cui si muovono e combattono i tank.

Si tratta dunque di un elemento molto importante dal punto di vista delle strategie di gioco. È possibile trovare mappe diversissime tra loro, soprattutto a fare la differenza sono le texture utilizzate che possono essere standard o custom, sia perché è possibile generarle casualmente tramite l'applicazione server, sia perché è possibile crearne di personalizzate con un apposito editor.

Nelle mappe custom è possibile inserire un elevato numero di varianti che rendono la grafica davvero notevole, varianti la cui valutazione può essere molto importante ai fini della vittoria.

Una mappa consiste in uno spazio 3D cubico in possono essere posizionati diversi oggetti. Ostacoli = si dividono a loro volta in muri, box (parallelepipedi), piramidi, e personalizzati. Importantissimi ai fini del gioco e non solo per fini estetici (soprattutto se uniti all'uso delle texture). Gli ostacoli di norma (eccettuati i poteri straordinari guadagnabili con alcune flag) non possono essere attraversati né dai tank né dai proiettili, così come non vi si vede attraverso e sono quindi fondamentali per sfuggire, nascondersi, tendere agguati ai tank avversari. Inoltre su alcuni di essi è possibile salire sopra, altri possono avere proprietà speciali come distruggere al tocco un tank (il “mare” nella Fig. 5.2 distrugge i tank) o esercitare su di esso delle forze (il corso d'acqua nella Fig. 5.1 trascina secondo corrente i tank che vi sono immersi).

Portali = oggetti di forma diversa, posizionati e collegati casualmente nella mappa, ma tutti con la medesima caratteristica, ovvero quella di trasportare (più o meno istantaneamente) il tank che ne usa uno in un altro punto della mappa; anche i proiettili passano attraverso i portali e questo permette di colpire tank molto lontani una volta scoperti i collegamenti tra portali.

Flag = bandiere posizionate casualmente nella mappa, si dividono in 3 tipologie: positive, negative e “di Team”. Le prime assegnano al tank che le raccoglie dei vantaggi come armi più potenti o telecomandate, la possibilità di passare attraverso gli Ostacoli, l'invisibilità, od altro; le flag negative al contrario assegnano un handicap temporaneo al tank come l'impossibilità di sterzare verso destra, l'impossibilità di saltare o il radar fuori uso; infine

le flag “di Team” sono usate soltanto nella modalità di gioco *Capture the Flag* e costituiscono appunto l’oggetto da rubare e portare alla propria base. Un tank può avere una sola flag per volta e quando viene distrutto rilascia la flag che resta nel luogo in cui è esploso.

Data l’importanza di tutti questi fattori è evidente come la mappa sia fondamentale nello studio di strategie di gioco, per questo è stata creata la classe *NetworkMapper* atta alla lettura dei dati della mappa ed alla sua trasformazione in una matrice 2D che poi può venir salvata all’occorrenza su file.

La classe, una volta costruitone l’oggetto ed attivato il metodo *Mapper*, legge i dati degli oggetti grafici (come piramidi, box, ecc.). Questi dati vanno poi convertiti dal sistema di coordinate cartesiane a quello matriciale ed a questo punto è possibile utilizzarli per riprodurre su una matrice l’oggetto. La matrice viene inizializzata tutta a 0, è sempre una matrice quadrata e le dimensioni sono variabili a seconda della grandezza della mappa.

Per ogni ostacolo da riprodurre sulla mappa viene creata una matrice temporanea e con l’algoritmo di Bresenham vengono disegnati su questa nuova matrice i contorni dell’ostacolo.

L’algoritmo di Bresenham utilizzato è una versione perfezionata che permette di calcolare i punti di qualsiasi tipo di retta, a prescindere dal suo coefficiente angolare (la versione comune del Bresenham non permette di tracciare rette con coefficiente angolare negativo).

Una volta creata la matrice temporanea con rappresentato il perimetro dell’ostacolo specifico, tramite un algoritmo di Flood-Fill viene riempito l’interno del perimetro. La chiamata al metodo *FloodFill* avviene all’interno di un blocco *try* in modo da intercettare eventuali eccezioni sollevate, in questo caso viene stampata su file la matrice temporanea anche se incompleta, e si procede normalmente ma senza portare a termine il Flood-Fill per quell’ostacolo.

Eseguito il Flood-Fill la matrice temporanea viene ricopiata su quella permanente che raffigura tutta la mappa e viene liberata la memoria occupata dalla matrice locale.

Quando questo procedimento è stato eseguito per tutti gli ostacoli, la mappa globale viene salvata su file in cui lo spazio vuoto significa che quel punto sulla mappa è libero, gli “0” disegnano gli ostacoli e le “A” simboleggiano eventuali errori nel calcolo di quel punto.

Ovviamente sono presenti diversi controlli a vari livelli per evitare la scrittura su spazio non allocato dalla matrice globale.



Fig. 5.1 – Esempio di mappa con texture: se un tank entra nel corso d’acqua viene trascinato da una corrente e subisce la sua forza acceleratrice.



Fig. 5.2 - Esempio di mappa con texture: se un tank case in acqua viene distrutto.

5.3 Memorizzare le azioni

Nell'ottica dell'apprendimento può essere fondamentale per addestrare un agente ad eseguire prestabilite strategie, poter disporre delle azioni più comuni legate alle strategie.

In BZFlag le azioni più frequenti sono i movimenti, il salto e il fuoco, ma ovviamente a seconda dell'ordine e della frequenza con cui vengono eseguiti lo stile di gioco cambia radicalmente.

Sono stati implementati alcuni metodi della classe *NetworkInterface* ed una struttura che permettono per l'appunto di intercettare le azioni eseguite dal giocatore e di memorizzarle su di un file per diversi usi futuri.

All'attivazione dell'opzione di memorizzazione delle azioni di gioco, viene settato un "tempo zero" a partire dal quale vengono poi calcolati i tempi di tutte le azioni ed inizia l'intercettazione di tutte le key (i tasti della tastiera) premuti.

```
struct azione
{
    long tempo;
    int cod;
    bool press;
};
```

Tab. 5.1 – Struttura azione.

La struttura azione permette di memorizzare i dati utili legati alle azioni:

- tempo = centesimi di secondo passati dall'inizio dell'intercettazione; si tratta di un dato importantissimo perché indica la durata ed il momento di esecuzione di un'azione, e sono proprio questi insieme all'ordine delle azioni a stabilire la linea di gioco.
- cod = codice numerico identificativo dell'azione eseguita; permette di sapere di quale azione si tratta.

CAPITOLO 5

- `press` = variabile inserita esclusivamente per scopi di ottimizzazione del codice, infatti consente di evitare la memorizzazione di azioni uguali contigue (invece di essere segnata l'azione "6" al secondo "3" ed al secondo "4", viene segnata l'azione "6" dall'istante "3" fino a che non cambia tipo di azione).

Ogni azione viene poi aggiunta ad una *coda* (queue), e nel caso in cui il codice dell'azione corrisponda alla pressione del tasto "Esc" tutte le azioni memorizzate nella *coda* vengono scritte su di un file.

----- DATA: 02/09/06 ORA: 15:35:23 -----		
0	83	0
25	93	0
108	6	1
133	5	1
179	5	0
194	102	1
203	70	0
222	4	1
247	4	0
329	102	1
342	70	0
417	4	1
442	4	0
467	100	1
489	68	0
629	98	1
644	66	0
701	6	0
736	27	1

Tab. 5.2 – Esempio di output delle azioni su file: nella prima colonna i millisecondi, nella seconda il codice azione e della terza lo stato del tasto.

5.4 Eseguire le azioni

Una delle possibili applicazioni del Machine Learning a BZFlag starebbe nella scelta da parte dell'agente di una strategia fra tante prestabilite (ad esempio esplorare piuttosto che attaccare o scappare) e già "scritte".

Per permettere questo sono stati aggiunti alcuni metodi alla classe *NetworkInterface* che permettessero per l'appunto di leggere da un file scelto le azioni che il tank deve eseguire.

Il metodo *InitInputFile* alloca queue e threads per ogni file di input disponibile.

Ogni file viene scorso fino al termine (EOF) ed ogni riga del file viene memorizzata nella coda corrispondente utilizzando la struttura azione già vista.

Ogni coda sarà quindi corrispondente ad un file di input e quindi ad una strategia o comunque una serie di azioni di gioco.

I file di input per poter essere letti devono essere presenti nella stessa cartella dell'applicazione e devono avere il nome composto da *prefisso* + *numero* + ".txt" :

dove *prefisso* è il contenuto della costante BZ_INPUT (default = "bzflag_in");

e *numero* è un numero compreso fra 0 e 9 corrispondente all'identificativo del file (e poi della queue), la numerazione dei file deve essere progressiva e continua (non può esserci il file 5 senza il 4).

Con l'inizializzazione delle code e la lettura delle azioni da file si attiva la possibilità di far eseguire al tank le azioni corrispondenti ad una specifica queue.

Tramite la pressione del tasto numerico corrispondente alla coda (da 0 a 9) si attiva il thread corrispondente e si esegue la funzione *PlayActions* che scorre la coda delle azioni fino ad esaurirla. Per ogni azione memorizzata nella coda, si legge il tempo a cui deve essere eseguita e tramite la funzione *Sleep* dei thread si attende l'istante preciso dell'azione; una volta giunto il momento viene simulata la pressione della key corrispondente all'azione chiamando la funzione *doKey* e passandole un evento generato appositamente.

Quando tutte le azioni di una coda sono state eseguite, e quindi la coda è vuota, il thread corrispondente viene distrutto.

L'uso dei thread e della funzione *Sleep* permette l'esecuzione delle azioni al momento prestabilito senza dover tener bloccato il controllo del flusso e delle risorse mentre la funzione attende il momento per eseguire l'azione.

```

<linea> = <tempo> <cod> <press>
<tempo> = <long>
<cod> = <int>
<press> = <bool>

```

Tab. 5.3 – Formato dati dei file di input delle azioni.

5.5 Data Tank

Dati fondamentali per l'analisi e l'elaborazione di strategie sono quelli relativi ai tank, come per esempio la loro posizione, la direzione, la velocità, ecc.

A partire da questi dati potrebbe poi essere possibile capire cosa sia più conveniente fare (attaccare, nascondersi, ecc), in che direzione muoversi ed altre automazioni del genere.

```

struct dataTank
{
    const float* position;
    const float* velocity;
    float angularVelocity;
    float angle;
    TeamColor team;
    int distance;
    int trackingDiff;
    int label;
};

```

Tab. 5.4 – Struttura *dataTank*.

È stata creata la struttura *dataTank* i cui campi sono atti a memorizzare i dati importanti di ogni tank:

- o position = vettore di 3 elementi contenenti le coordinate tridimensionali che indicano la posizione del tank;

- velocity = vettore di 3 elementi contenenti le 3 componenti della velocità di movimento del tank;
- angularVelocity = velocità di rotazione del tank;
- angle = angolo indicante in quale direzione sta puntando il tank (e quindi direzione in cui si muove);
- team = colore indicante la squadra di appartenenza del tank;
- distance = distanza tra il tank del giocatore ed ognuno degli altri tank, calcolata tramite il Teorema di Pitagora.
- trackingDiff = differenza di puntamento ovvero l'angolo compreso fra la direzione di puntamento degli altri tank e la retta che li congiunge al tank del giocatore; in altre parole la differenza tra l'angolo corrispondente al coefficiente angolare della retta che passa per i due tank e l'angolo indicante la direzione di puntamento del tank avversario.

Tramite la pressione di un apposita key si avvia il metodo *SaveDataTank* che in modo dinamico alloca le strutture necessarie per memorizzare i dati di tutti i tank.

Dopo aver terminato il calcolo di tutti i dati per ogni tank presente in game, tali dati vengono scritti dai vettori in cui sono memorizzati ad un file.

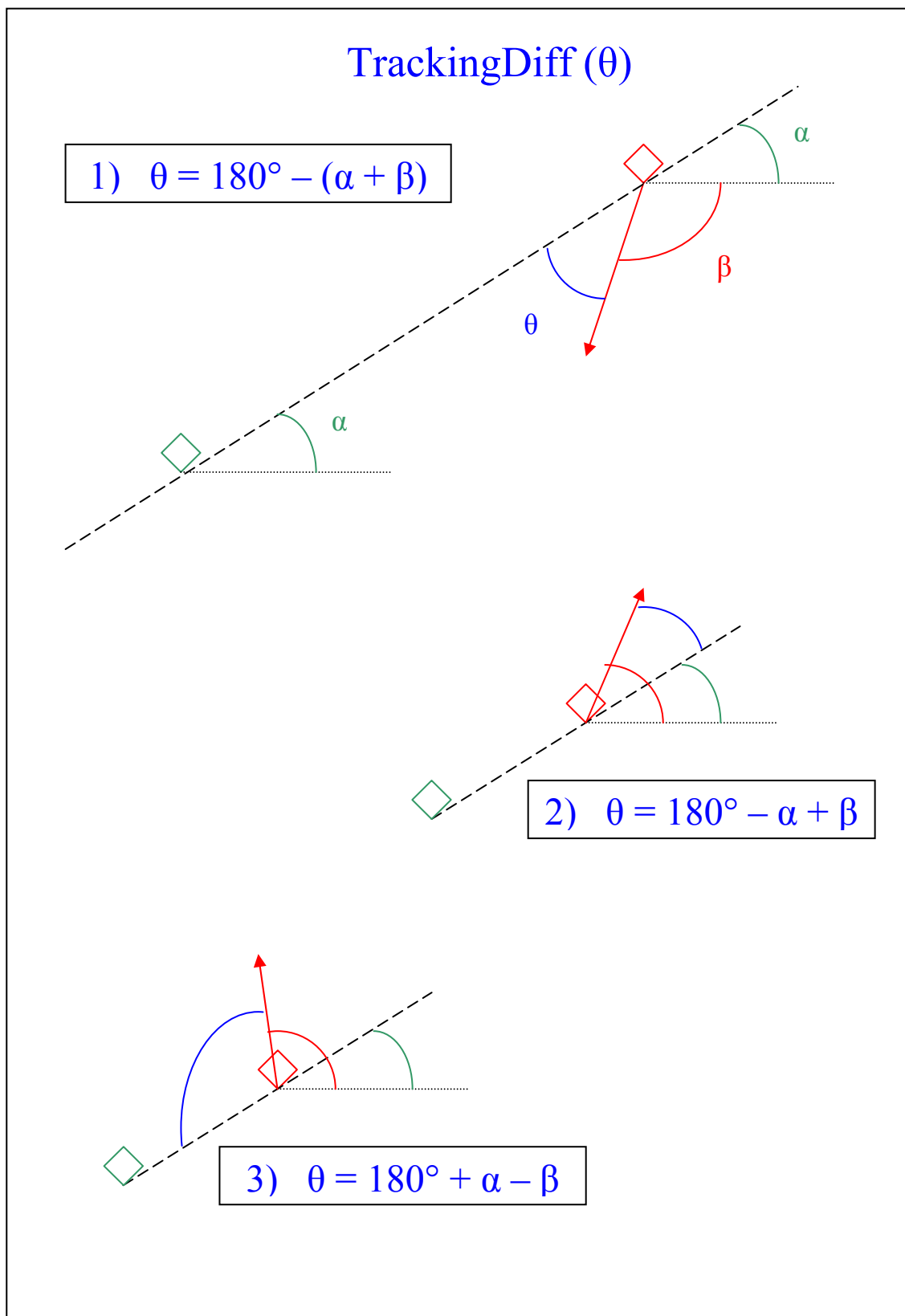


Fig. 5.3 – Alcuni casi di calcolo della “Differenza di puntamento”.

----- DATA: 02/10/06 ORA: 13:01:17 -----			
DATI GIOCATORE COMPUTER			
Posizione :	-487.977	507.085	161.607
Velocità :	36.4378	39.7433	-31.7631
Velocità Angolare:	-0.785344		
Direzione:	6.21557		
Team:	3		
DATI GIOCATORE 0			
Posizione:	740.974	-1.88697	30
Velocità:	-39.9492	2.00363	0
Velocità Angolare:	-0.0402759		
Direzione:	3.09077		
Team:	1		
Distanza:	1330		
Differenza Puntamento:	19		
DATI GIOCATORE 1			
Posizione:	-975.274	25.1017	32.0772
Velocità:	38.9383	9.15672	-7.4751
Velocità Angolare:	-0.785398		
Direzione:	5.93242		
Team:	3		
Distanza:	685		
Differenza Puntamento:	65		
DATI GIOCATORE 2			
Posizione:	-1075.89	-27.6575	30.0005
Velocità:	-6.33513	-18.9705	0
Velocità Angolare:	0.785398		
Direzione:	1.28161		
Team:	1		
Distanza:	794		
Differenza Puntamento:	31		

Tab. 5.5 – Esempio di output su file dei dati dei tank in gioco.

5.6 Controllo nemici

Vera e propria parte di Machine Learning di questo progetto è il *controllo dei nemici* che consiste nella capacità di saper individuare fra i tank avversari quelli che possono essere pericolosi per il giocatore.

Per questa funzionalità vengono usate le Support Vector Machine tramite la libreria LIBSVM.

È possibile individuare due fasi nell'uso delle SVM per i fini sopradetti:

1. training;
2. predict.

5.6.1 Training

La fase di *training* consiste nel raccogliere i dati necessari all'addestramento e nel passarli, forniti di etichette, alla libreria LIBSVM che procede a creare (secondo i parametri passati) un *modello* utilizzabile alla risoluzione del problema anche con altri set di dati, ovvero una generalizzazione della soluzione al problema.

Il problema è quello di riconoscere i tank pericolosi ed è stato deciso di effettuare questa valutazione in base a due parametri:

- distanza;
- differenza di puntamento.

Una volta attivata l'opzione di raccolta dei dati, tramite le funzioni *TrainThread* e *GetTrainingData* viene creato un thread che ad intervalli regolari (default = 10 sec) invoca il metodo *SaveDataTank* che calcola *distanza* e *differenza di puntamento* per ogni tank avversario e li valuta.

La valutazione di tali parametri consiste del controllare se la distanza o l'angolo della differenza di puntamento sono inferiori alle costanti `DISTANCE_MIN` e `ANGLE_MIN`.

Se la distanza è inferiore alla costante significa che il nemico in questione è entro una certa distanza dal tank del giocatore e quindi potrebbe essere pericoloso (un tank molto lontano può essere ugualmente pericoloso ma è meno probabile); se invece l'angolo è inferiore alla costante apposita significa che il nemico sta puntando un punto vicino al tank del

giocatore e questo può ovviamente indicare che sta per fare fuoco sul giocatore o che vuole avvicinarsi, ipotesi entrambe pericolose.

Nel caso in cui una delle suddette condizioni sia vera allora ai dati in input viene assegnata etichetta “+1”, nel caso opposto etichetta “-1”.

I dati etichettati vengono salvati in un file e quando sono in numero sufficiente è possibile passare i dati in esso contenuti alla funzione *svm_train* della libreria LIBSVM insieme agli opportuni parametri, e la funzione procede con l’addestramento della SVM fino a generare un modello di soluzione generalizzata del problema.

1	1:563	2:14
-1	1:284	2:174
-1	1:482	2:35
-1	1:261	2:173
-1	1:431	2:21
-1	1:309	2:177
-1	1:198	2:62
1	1:132	2:22
-1	1:639	2:26
-1	1:660	2:83
1	1:233	2:3
-1	1:548	2:106
1	1:94	2:3
1	1:310	2:15
-1	1:554	2:175
1	1:131	2:2
-1	1:401	2:31

Tab. 5.6 – Esempio di dati di addestramento per la LIBSVM.

Parametri di *svm_train*:

- ❖ -s 0 => C-SVM
- ❖ -t 1 => polinomial kernel
- ❖ -d 2 => degree = 2
- ❖ -m 100 => memoria cache = 100MB

Tab. 5.7 – I parametri usati nell’addestramento dell’agente con LIBSVM.

```

*
optimization finished, #iter = 86
nu = 0.551567
obj = -100.542538, rho = 1.638957
nSV = 121, nBSV = 116
Total nSV = 121

```

Fig. 5.4 – Dati forniti dalla LIBSM circa l’addestramento appena eseguito.

```

svm_type c_svc
kernel_type polynomial
degree 2
gamma 0.5
coef0 0
nr_class 2
total_sv 16
rho -939.896
label 1 -1
nr_sv 8 8
SV
1 1:142 2:61
1 1:427 2:17
0.1611393639333437 1:138 2:149
0.5161964476205758 1:458 2:19
1 1:137 2:70
1 1:604 2:17
1 1:135 2:136
0.5795457885972273 1:120 2:70
-1 1:431 2:21
-0.06938612732236148 1:639 2:26
-0.187495472828385 1:445 2:44
-1 1:182 2:22
-1 1:607 2:23
-1 1:151 2:176
-1 1:151 2:27
-1 1:160 2:27

```

Tab. 5.8 – Esempio di file “.model” contenente il modello generato con l’addestramento.

5.6.2 Predict

Una volta in possesso del modello della soluzione al problema è possibile, disponendo dei dati di *distance* e *trackingDiff*, generare una predizione della minaccia costituita da ogni tank.

Questo grazie alla funzione *svm_predict* della LIBSVM che riceve in input il modello della soluzione ed un vettore della struttura *svm_node* i cui campi sono riempiti con i dati di *distance* e *trackingDiff*.

La funzione restituisce in output TRUE se una delle due condizioni impostate è vera e quindi se il tank avversario è molto vicino a quello del giocatore o sta puntando all'incirca nella direzione in cui si trova il giocatore, restituisce FALSE se nessuna condizione è verificata.

L'output della funzione è ovviamente utilizzabile per segnalare al giocatore quali sono gli avversari pericolosi in real-time, man mano che le situazioni di gioco evolvono.

5.6.3 Test e risultati

Il “Controllo di Minaccia”, specialmente la versione automatica, è stata lungamente testato in rete. Essendo il gioco free, multiplayer e piuttosto diffuso non è stato difficile eseguire numerosi test.

Dai test emerge chiaramente che l'agente intelligente funziona perfettamente e riesce sempre a distinguere i casi minacciosi dagli altri secondo quanto volevamo insegnarli, ovvero quando sono troppo vicini (distanza < 150) o stanno puntando vicino a noi (differenza di puntamento $< 20^\circ$); questo perché si tratta di un problema linearmente separabile.

D'altra parte risulta chiaro che l'agente addestrato non si dimostra utile in tutti i casi: infatti quando il numero di giocatori è piuttosto elevato (più di una decina di *tank*) le possibilità che un tank punti nella nostra direzione per caso o ci passi vicino ma senza essere interessato a noi sono abbastanza alte da rendere inefficace il nostro sistema.

Allo stesso modo nei casi con soltanto 2 o 3 giocatori in tutto è scontato che gli avversari cerchino di distruggere proprio il nostro *tank*.

Al contrario quando ci troviamo in mondi a media popolazione il “Controllo” diviene molto utile perché ci permette di focalizzare l'attenzione su tank, anche a grande distanza, che puntino su di noi.

CAPITOLO 5

Il nostro agente si dimostra poi davvero rivoluzionario nel controbattere all'uso di alcune flag da parte degli avversari. Una di queste flag è la “Cloak” che rende il tank invisibile sulla schermata principale ma non sul radar, in questo caso ovviamente il “Controllo di Minaccia” ci segnala che il nemico ce l'ha con noi anche se noi non lo vediamo, dunque la nostra attenzione viene attratta dal cambio di colore sul radar e possiamo stare più attenti. Un'altra flag è la “Super Bullet” che permette ai proiettili sparati dal tank di attraversare gli ostacoli: la più comune ed efficace strategia per chi detiene questa flag è quella di nascondersi dietro al maggior numero di ostacoli possibile e sparare da lì mirando non sulla visuale principale ma tramite il radar, questo fa sì che in genere il tank che spara non sia nei pressi delle vittime che vengono distrutte senza quasi accorgersene. Ovviamente il “Controllo” permette di essere avvisati in questi casi e ci toglie dalla condizione di “vittime inconsapevoli” dandoci la possibilità di spostarci in tempo e prendere le adeguate contromisure.

Come già detto il nostro agente si dimostra inefficace in diverse situazioni e bisogna ammettere che ciò che viene fatto tramite esso potrebbe essere fatto anche senza l'utilizzo di tecniche di Machine Learning. C'è però da considerare che sarebbe possibile potenziare notevolmente l'agente, rendendolo un'utile ausilio in un maggior numero di casi, aggiungendo altre caratteristiche (features) a quelle ora usate per la valutazione della minaccia (distanza e differenza di puntamento). L'aggiunta di features come la velocità o il tipo di flag eventualmente detenuta dai tank potrebbero senza dubbio rendere l'agente molto superiore al livello attuale. Grandi benefici si potrebbero ottenere anche se si volesse aggiungere un fattore temporale, infatti se si potesse valutare le situazioni come sequenze e non come istantanee si guadagnerebbe notevolmente in precisione: Un *tank* può puntare verso di noi ad un certo istante perché vuole spararci o casualmente perché sta muovendo, se dopo qualche secondo ci sta ancora puntando le probabilità che inseguia proprio noi sono ovviamente più alte, dunque il considerare sequenze invece che istanti renderebbe più accurato e preciso il “Controllo di Minaccia”.

Simili miglioramenti sarebbero indubbiamente utili ma al contempo renderebbero la valutazione troppo complessa per algoritmi non facenti uso di Machine Learning, al contrario il nostro sistema è già completamente predisposto all'aggiunta di simili caratteristiche e l'uso delle SVM renderebbe la loro gestione facile ed efficiente.

Conclusioni e sviluppi futuri

Tutte le funzionalità che si aveva intenzione di aggiungere al videogioco BZFlag sono state aggiunte con successo e con ottimi risultati.

In special modo il Controllo Nemici presenta un'utilità evidente ed immediata per il giocatore, e test svolti su tale funzione dimostrano come l'apprendimento con le SVM sia andato a buon termine e l'agente abbia pienamente imparato a distinguere una situazione minacciosa. La fase di addestramento, che spesso è piuttosto lunga, è stata eseguita in tempi molto brevi grazie soprattutto alla grande facilità con cui sono stati raccolti i dati necessari. Il fatto che si tratti di un gioco multiplayer e free rende infatti piuttosto semplice e veloce il reperire dati sia per le fasi di studio ed analisi, di addestramento e di test; e proprio questi fattori potranno in futuro far sì che si ottengano molti risultati procedendo in questa direzione.

Infatti le features che abbiamo aggiunto a BZFlag guadagnano importanza soprattutto se viste in un'ottica di incompletezza del progetto. Così come era stato supposto e programmato non si è riusciti a sviluppare appieno le potenzialità del Machine Learning per questo gioco, ma si è predisposto l'ambiente a sviluppi futuri e sono state date le prime dimostrazioni (con il "Controllo di Minaccia") di cosa sia possibile sviluppare.

La maggior parte delle nostre aggiunte guadagneranno importanza soprattutto in futuro quando potranno permettere con sforzi e tempi ridotti di continuare lo studio e l'applicazione di tecniche di AI a questo gioco.

CONCLUSIONI E SVILUPPI FUTURI

Infatti come era stato premesso in questa tesi si è voluto in primo luogo constatare le effettive possibilità di applicazioni di Machine Learning a un gioco simile, ed in secondo luogo si è voluto gettare la basi per rapidi sviluppi futuri, rendendo il sistema in grado di interagire con agenti intelligenti e di fornire i dati necessari.

I prossimi sviluppi di questo progetto potrebbero generare agenti intelligenti in grado di:

- effettuare un “Controllo di Minaccia” efficace in un maggior numero di casi;
- suggerire al giocatore strategie con cui opporsi a nemici in possesso di flag (poiché le flag modificano radicalmente il gioco, le strategie consuete possono non essere adatte);
- cercare la flag più vantaggiosa per un determinato scopo o stile di gioco;
- esplorare autonomamente e quindi muoversi senza il giocatore;
- inseguire un avversario scelto dal giocatore;
- evitare i nemici;
- evitare i proiettili avversari, questo pur lasciando il controllo del movimento al giocatore;
- fare fuoco quando un tank è a portata di tiro e ci passa innanzi (lasciando al giocatore il solo compito di muovere il tank);

e queste sono soltanto alcune delle possibilità, più o meno vicine.

La connotazione opensource dell'applicazione è sicuramente uno dei suoi massimi punti di forza e permetterebbe senza dubbio sviluppi futuri nel campo dell'AI come in molti degli altri campi dell'informatica (grafica 3D, database, tecnologie di internet, ecc.).

Unica nota dolente del progetto è la scarsa documentazione disponibile e l'assenza di commenti nel codice che rendono molto più lunga e complessa la comprensione degli algoritmi e dei metodi presenti.

Bibliografia

- [1] K. Stanley, B. Bryant, R. Miikkulainen. *Evolving Neural Network Agents in the NERO Video Game*. Winner of CIG 2005 Best Paper Award at IEEE Symposium on Computational Intelligence and Games 2005.
- [2] Nick Palmer. *Machine Learning in Games Development*.
URL <http://ai-depot.com/GameAI/Learning.html> .
- [3] Mat Buckland. *Programming Game AI by example*. Wordware Publishing, Inc. 2005.
- [4] S. Cacciaguerra, M. Roffilli: *Agent-based participatory simulation activities for the emergence of complex social behaviours*. Proc. of AISB05, Social Intelligence and Interaction in Animals, Robots and Agents, April 12-15, 2005, Hatfield, England.
- [5] S. Cacciaguerra, A. Lomi, M. Roccetti, M. Roffilli, *A Wireless Software Architecture for Fast 3D Rendering of Agent-Based Multimedia Simulations on portable Devices* , IEEE Consumer Communications and Networking Conference 2004, Caesars Palace, Las Vegas, Nevada, USA, 5-8 January 2004.

BIBLIOGRAFIA

- [6] S. Cacciaguerra, S. Ferretti, M. Roccetti, M. Roffilli: *Car Racing through the Streets of the Web: a High-Speed 3D Game over a Fast Synchronization Service* to be presented at WWW2005, May 10-14, 2005, Chiba, Japan.
- [7] M. Roffilli. *Advanced Machine Learning Techniques for Digital Mammography*. PhD thesis, University of Bologna, Department of Computer Science, 2006.
- [8] BZFlag website: URL <http://www.bzflag.org> .
- [9] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer Verlag Inc. New York, 1995.
- [10] V. Vapnik. *Statistical Learning Theory*. J.Wiley and Sons Inc. New York, 1998.
- [11] V. Maniezzo. *Support Vector Machine*. University of Bologna.
URL <http://www3.csr.unibo.it/~maniezzo/didattica/SoftComputing/SVM.pdf>
- [12] LIBSVM website: URL <http://www.csie.ntu.edu.tw/~cjlin/libsvm> .
- [13] Pthreads website: URL <http://sourceware.org/pthreads-win32> .