UNIVERSITÀ DI PISA
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Triennale in Informatica

RELAZIONE DI TIROCINIO

Caratterizzazione del contenuto del traffico di rete

RELATORE
Luca DERI

CORRELATORE
Daniele SARTIANO

Candidato
Michele CAMPUS

ANNO ACCADEMICO 2016-2017

ringraziamenti

Indice

Capitolo 1

Introduzione

Internet, fin dalla propria nascita, è in costante evoluzione. Questa trasformazione ha reso necessario lo sviluppo di tecniche per capire la tipologia di informazione che viaggia in rete, puntando a garantire anche una maggiore sicurezza. Se fino a poche decine di anni fa gli *host* connessi in rete erano soprattutto server e computer aziendali, la diffusione capillare di Internet, andata di pari passo con il crescente numero di dispositivi mobili, ha reso necessario il monitoraggio del traffico *web*, sia per il privato cittadino che per le aziende. Il continuo sviluppo di applicazioni *web* e la crescita di dati multimediali accessibili a qualsiasi utente, ha trasformato la caratterizzazione del traffico, rendendola una vera e propria sfida. L'aumento delle applicazioni di rete ha portato con se' lo sviluppo di nuovi protocolli; il primitivo metodo di classificazione basato sulla corrispondenza *porta = protocollo*, non ha più alcun senso se non coadiuvato da tecniche più complesse e affidabili. Se prima eravamo interessati al riconoscimento di un protocollo esclusivamente a livello di rete, ora è fondamentale classificare il traffico a livello applicativo: non ci interessa sapere solo se ci sono dati HTTP(S) o FTP, ma, ad esempio, se quel flusso non sia Facebook piuttosto che Youtube.
Per fare ciò si utilizza la tecnica del Deep Packet Inspection (DPI): si cerca di classificare un pacchetto in base all'analisi della propria parte dati, estraendo i *metadati* che ci permettono di ottenere le informazioni del flusso a livello applicativo.

Il Deep Packet Inspection ci dà l'idea di come classificare al meglio il traffico a livello applicativo dal punto di vista dei protocolli. Tuttavia non abbiamo alcuna percezione del tipo di dato che sta viaggiando in rete, soprattutto per protocolli generici come HTTP. La grande quantità di informazioni multimediali che oggi naviga su internet rimane esclusa dalla classificazione.

Vedendo il problema da punto di vista più alto, si è pensato di aggiungere un livello di caratterizzazione dei dati multimediali (immagini) che viaggiano sulla rete, utilizzando la tecnica del Machine Learning, in modo da complementare il lavoro delle librerie di DPI. L'idea è quella di estrarre l'immagine

e sottoporla ad un modello di classificazione costruito ad hoc. Per raggiungere tale obiettivo è necessaria l'implementazione di una rete convoluzionale, cioè una rete neurale che utilizza il processo di trasformazione dell'immagine e la associa ad una classe predefinita.

Con le tecniche di DPI, da un pacchetto HTTP si estrae l'`url` del sito, si scaricano i byte dell'immagine associata e si fa una classificazione "al volo" di tale immagine grazie alla rete neurale. Se il contenuto dell'immagine è illecito, il sito viene inserito in una *blacklist*.

Le immagini sono classificate come **Lecite** o **Illecite**. Questo permette ad applicazioni di monitoraggio di implementare politiche di rete basate anche sul filtraggio dei contenuti non appropriati.

1.1 Classificazione del traffico di rete e caratterizzazione dei contenuti

Fino a pochi anni fa il traffico Internet era prevalentemente basato sulla visita di pagine web, scambio mail o file. Adesso è quasi impensabile concepire un'operazione che non preveda l'accesso ad una rete, pubblica o privata che sia. Pensiamo ai movimenti bancari, agli acquisti su Internet, alla telefonia, ma anche televisori o elettrodomestici. Questa vera e propria esplosione ha portato un incremento sia degli host connessi in rete che delle applicazioni Internet. La classificazione del traffico di rete si basa proprio sulla necessità di capire qualitativamente che tipo di applicazioni e protocolli stanno viaggiando in una rete.

1.1.1 Deep Packet Inspection

Per questo *task* è necessaria la classificazione del traffico a livello applicativo; è richiesta l'analisi del payload per estrarre informazioni caratterizzanti il protocollo. Non è più sufficiente l'analisi del pacchetto a livello TCP/IP come poteva essere fino a qualche anno fa, né tantomeno l'associazione porta X = protocollo Y.

Per far fronte a queste problematiche ha la sua genesi la tecnica del **Deep Packet Inspection (DPI)** [7]. A differenza delle classiche tecniche di monitoraggio basate sulla quintupla ≪*Src IP, Dst IP, Src Port, Dst Port, L4 Protocol*≫, il DPI esamina l'intero *payload* del pacchetto per determinare il protocollo a livello applicativo. In molti casi l'identificazione dei protocolli è un confronto tra il *payload* del pacchetto e un *pattern* noto per quel protocollo (se il protocollo è conosciuto), oppure è la ricerca di caratteristiche che specifichino tale protocollo. Con questa tecnica, vecchi e nuovi protocolli possono essere riconosciuti con la massima accuratezza. Un primo problema nasce quando il traffico da classificare è rappresentato da protocolli applicativi che non hanno chiari pattern su cui poter fare DPI e fingono di essere

altri: *Skype* è un emblematico esempio, poiché genera molti falsi positivi per il protocollo HTTP.

Il limite più grande è rappresentato dai protocolli criptati. Non potendo effettuare DPI sul payload cifrato, si devono percorrere strade parallele; a fronte di una minore accuratezza nel numero di flussi riconosciuti, l'estrazione degli *name server* da certificati SSL o *query* DNS è un ottimo metodo per poter classificare i protocolli cifrati, in quanto applicazioni come **Facebook, Twitter, Netflix** hanno i propri server.
Una cosa importante da sottilineare è che con il DPI non si tocca minimamente la parte sensibile dei dati.

1.1.2 Machine Learning

A causa del crescente numero di protocolli cifrati a livello applicativo, una tecnica sviluppatasi è quella della *classificazione statistica*, la quale non prevede il coinvolgimento di tecniche di DPI ma basa la propria euristica su parametri metrici di ogni protocollo grazie al **Machine Learning (ML)** [8]. Uno dei maggiori problemi dell'uso del Machine Learning per la classificazione dei protocolli è che per ottenere buoni risultati si necessita di un *trainig set* di ottima qualità, per far sì che il modello non possa apprendere in maniera errata; se istruiamo la rete con un dataset incongruente, esso imparerà in maniera incorretta ed avremmo risultati falsificati sulla classificazione dei protocolli.

Il Machine Learning non comporta grandi miglioramenti rispetto al Deep Packet Inspection dal punto di vista della classificazione dei protocolli.
Ecco perché è invece possibile sfruttarlo in modo differente.

Abbiamo detto che il DPI classifica i flussi in base ai protocolli e alle applicazioni, ma non ci dà alcuna possibilità di analizzare il contenuto del traffico che sta viaggiando. E' possibile invece effettuare questa operazione costruendo un modello di rete neurale che caratterizzi i dati (immagini) proveninenti da traffico non cifrato.

Si cerca di introdurre un nuovo livello di classificazione in modo da migliorare la grana del monitoraggio ed eventualmente bloccare il traffico non consentito.

Da qui l'idea di creare un prototipo per il lavoro complementare tra una libreria di Deep Packet Inspection e un modello di classificazione automatica basato su una rete neurale.

Struttura della relazione

- **Capitolo 2**. In questa sezione analizziamo l'attuale stato dell'arte sia per le librerie di Deep Packet Inspection, sia per i modelli di rete convoluzionale per la classificazione delle immagini. Per il DPI si evidenziano i pro e i contro di alcune librerie esistenti.
- **Capitolo 3**. Si parla di una libreria utilizzata, nDPI, che è stata estesa con il riconoscimento di un nuovo protocollo, illustrandone il funzionamento.
- **Capitolo 4**. Si introducono i concetti chiave del Machine Learning, in particolare del Deep Learning, i quali ci consentono la creazione del modello di rete neurale.
- **Capitolo 5**. Si mostra l'implementazione della rete neurale convoluzionale grazie all'utilizzo della libreria Tensorflow.
- **Capitolo 6**. Verranno evidenziati i risultati ottenuti, al fine di validare il modello implementato. Si espone l'idea di un prototipo di un framework per l'estensione di nDPI con la rete neurale creata, illustrando un caso d'uso reale.

Capitolo 2

Stato dell'Arte

In questo capitolo affronteremo due diversi stati dell'arte, per il Deep Packet Inspection prima e per il Deep Learning dopo.

2.1 Metodologie per classificare il traffico di rete

Questa sezione descrive lo stato dell'arte per quanto riguarda le tecniche utilizzate per la classificazione del traffico. Il problema dell'identificazione dei protocolli è in continuo cambiamento a causa dell'opposto lavoro tra chi vuole mantenere l'anonimato riguardo la propria applicazione, e chi vuole dare un'identità al traffico, caratterizzandolo e rendendolo monitorabile.

2.1.1 Tecniche di classificazione

Port based Come evidenziato dalla tabella sottostante, ad ogni protocollo è assegnata una porta di default. Con questo metodo si tenta di classificare il traffico grazie all'associazione *porta/protocollo*.

Port	Protocol
20	FTP data
21	FTP control
22	SSH
23	Telnet
25	SMTP
53	DNS
80	HTTP
111	RPC
443	HTTPS (SSL/TLS)

Tabella 1: classificazione porta protocollo definita da IANA[12]

L'utilizzo esclusivo di questa tecnica è ormai obsoleto, e comporta una bassa accuratezza dovuta al fatto che molte applicazioni internet non girano solo sulla porta standard associata al protocollo utilizzato. Su una singola porta passano più protocolli, non solo quello di default (ad esempio *Skype* utilizza la porta 80 come HTTP). In più c'è il problema che molte applicazioni utilizzano porte non associate, o un numero variabile di porte. Il protocollo RTP *(Real Time Protocol)*, ad esempio, ha un range di porte assegnate dalla 16384 alla 32767. L'unico vantaggio di questo metodo è il basso overhead dovuto al fatto che il metodo non prevede alcun salvataggio delle informazioni utili per la classificazione, visto che la porta può essere recepita al volo analizzando il pacchetto [13].

Packet based Ogni pacchetto è analizzato indipendentemente dagli altri. E' il principio su cui si basano software di analisi del pacchetto come *Wireshark* [16], i quali danno una classificazione in base a dissector che possano riconoscere il protocollo. Questa tecnica è molto precisa *sul pacchetto*, non dandoci però una visione delle applicazioni che stanno girando: è impossibile classificare Facebook, Youtube, Skype, Whatsapp etc., le quali richiedono approcci più complessi e costosi (in termini di risorse usate). E' una buona tecnica per analizzare il pacchetto ed estrarre informazioni utili.

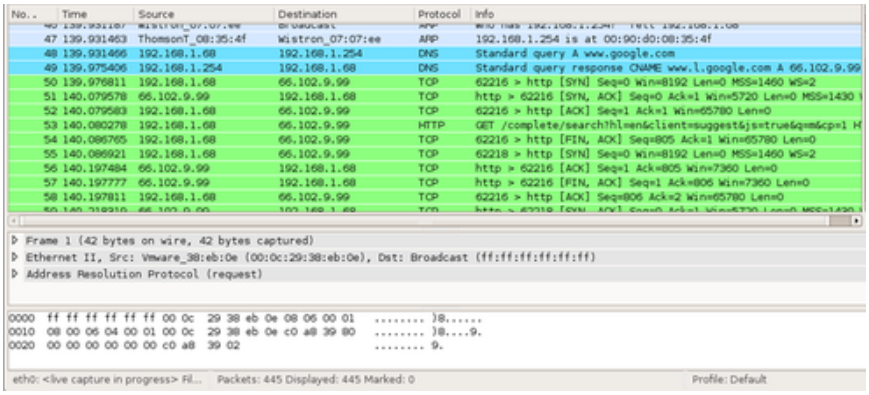

Figura 2.1: Esempio di analisi *packet based* con Wireshark

Flow based Un flusso è definito come una quintupla ⟨ Source IP address, Destination IP address, Source Port, Destination Port,

Transport Protocol ID ⟩. La classificazione avviene utilizzando le informazioni estratte dal singolo pacchetto e quelle accumulate per il flusso. Questo metodo porta notevoli benefici nell'accuratezza della classificazione a fronte di un maggiore consumo di risorse (le informazioni del flusso devono essere salvate temporaneamente o per un perioo di lunga durata). Con questa tecnica possiamo riconoscere protocolli che con l'analisi del singolo pacchetto sarebbe impossibile. L'introduzione del concetto di flusso ha portato il DPI ad essere molto più performante, unendo l'accuratezza delle informazioni ottenute analizzando il pacchetto, con quelle del flusso (per esempio quanti pacchetti di un certo tipo sono passati, se abbiamo estratto un certificato SSL, etc.). Inoltre i flussi rappresentano la base per la costruzione dei modelli di Machine Learning, in quanto questo metodo si basa su dati statistici e features ottenuti dall'analisi di una porzione di traffico, non dal singolo pacchetto.

Payload based E' il metodo su cui si basa il *Deep Packet Inspection (DPI)*. Questa tecnica cerca di dare una classificazione del protocollo analizzando la parte dati del pacchetto, grazie a specifiche caratteristiche del protocollo a livello applicativo che possono essere analizzate e confrontate tramite dissectors, ed eventualmente correlare i vari pacchetti dello stesso flusso. Si ricerca il protocollo ***vero***.

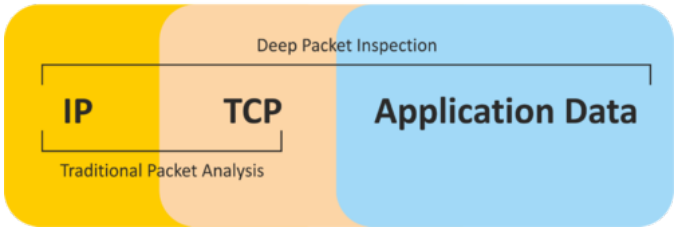

Figura 2.2: Analisi di un pacchetto con il DPI

Ogni dissector decodifica la parte del payload secondo le euristiche del proprio protocollo: se avviene la corrispondenza tra la parte analizzata e la parte di codice del dissector, il pacchetto (e quindi il flusso a cui appartiene) viene associato al protocollo, altrimenti, il controllo viene passato al successivo dissector, che farà la solita operazione.

L'analisi del livello applicativo permette di ottenere informazioni anche riguardo un certificato SSL o una query DNS, e poter riconoscere una determinata applicazione grazie all'host name contenuto. In questo modo possiamo (in parte) risolvere il problema della cifratura dei pacchetti a livello applicativo, potendo classificare flussi Facebook o YouTube grazie al nome del provider estratto.

Un simile approccio riguarda anche l'indirizzo IP. Estraendo e confrontando l'IP del pacchetto con un range di indirizzi noti per certi protocolli o

applicazioni, possiamo determinare la natura di tale pacchetto. Analizzare il payload quindi non esclude tuttavia le tecniche precedenti, anzi, cerca di sfruttarle per dare un'accuratezza di classificazione il più alta possibile.

Statistical based Si cerca di classificare il traffico in base alle informazioni riguardanti la distribuzione dei pacchetti, il tempo di interarrivo tra un pacchetto e l'altro, la dimensione o la durata di un flusso. Questa è la principale tecnica utilizzata per il training di un modello di machine learning; si costruisce un modello che possa essere in grado di caratterizzare i protocolli in base alle proprietà statistiche. A differenza del metodo stocastico, l'inferenza è data da fattori *esterni* alle specifiche informazioni contenute nel payload. Si possono distinguere due grandi gruppi di Machine Learning che vengono impiegati in questa metodologia:

- **approccio *supervisionato*** estrae la struttura delle informazioni per classificare le nuove istanze in classi predefinite. Il metodo si dice supervisionato perché le classi di output sono state definite in precedenza.
- **approccio *non supervisionato*** questo metodo non necessita di un dataset etichettato, perciò l'output del modello non classifica le istanze in classi predefinite. È il metodo stesso che cerca di raggruppare i dati in gruppi (cluster) in base a caratteristiche comuni.

2.1.2 Soluzioni software esistenti

Descriviamo ora alcuni lavori che riguardano la classificazione dei protocolli, analizzando le soluzioni a livello software e riportando anche i risultati ottenuti confrontando i vari tool [17].

Libprotoident

Libprotoident [18] è una libreria scritta in C che introduce il concetto di *Lightweight Packet Inspection (LPI)*, ovvero una forma di DPI che limita la ricerca del protocollo analizzando i primi quattro byte del payload applicativo. Il tool supporta diversi protocolli, e per ognuno di essi è sviluppato un dissector. Il riconoscimento avviene passando il payload del pacchetto ad una funzione che controlla se i primi quattro byte della parte dati corrispondono a quelli specificati nell'identificatore: se il match avviene, il protocollo è riconosciuto, altrimenti si passa il payload al successivo dissector, e il processo riparte. Libprotoident utilizza anche informazioni sulle porte e sugli indirizzi IP per ottenere la classificazione. In aggiunta i protocolli possono essere raggruppati in macro categorie rappresentanti la propria "natura" (peer-to-peer, mail, web application, ...). Il principale vantaggio di questa libreria è la minimalità delle risorse richieste e occupate, poiché interessano

solo quattro byte dell'intero payload. Lo svantaggio è però rappresentato da una minor accuratezza, soprattutto per quanto riguarda i protocolli su cui girano applicazioni che necessitano controlli su una parte più ampia del payload; si rischiano di avere molti pacchetti non riconosciuti o falsi positivi.

L7-filter

Questo classificatore è nato come supporto al progetto Netfliter [19] in versione kernel. Successivamente, nel 2006, è stata rilasciata anche la versione software a livello utente. Per la classificazione del protocollo utilizza espressioni regolari a livello applicativo. Il principale scopo di questa libreria è il riconoscimento di protocolli che usano porte impredicibili. In generale, L7-filter utilizza tre metodi per la classificazione del traffico:

- Identificazione numerica del pacchetto, utilizzando i moduli standard di ip-tables
- Espressioni regolari per determinare la natura del livello applicativo del pacchetto
- Funzioni per caratterizzare il traffico

L7-filter [20] definisce un set di file di espressioni regolari da poter utilizzare. Ad esempio per cercare la corrispondenza del protocollo "foo" l'utente deve utilizzare l'espressione regolare

```
iptables -m layer7 --l7proto foo
```

e iptables leggerà tale espressione che definisce "foo" in */etc/l7-protocols/*/foo.pat*. Purtroppo l7-filter tende a produrre diversi falsi positivi, soprattutto riguardo a protocolli peer-to-peer, poiché molto difficili da riconoscere con specifici pattern. Un altro problema riguarda il fatto che ogni espressione regolare deve essere scritta manualmente; ciò significa che ogni protocollo necessita di essere studiato in modo approfondito, cercando di astrarre e generalizzare le peculiari caratteristiche che consentano la scrittura di pattern precisi.

SPID

Un interessante algoritmo che ha dato vita a diversi software è lo Statistical Protocol Identification (SPID) [21]. Si basa sull'analisi statistica dei flussi, utilizzando il confronto probabilistico sul numero di pattern trovati all'interno del payload. Da questo punto di vista l'algoritmo SPID può essere considerato un ibrido tra il deep packet inspection e le tecniche di analisi statistica dei flussi. I cosiddetti *fingerprint* di ogni protocollo sono rappresentati come distribuzioni probabilistiche in cui si hanno due vettori divisi in cluster, uno per il conteggio dei pattern trovati durante l'analisi

del pacchetto, l'altro per le probabilità, intese come la normalizzazione del vettore dei contatori.

SPID ha però un limitato numero di protocolli che può catalogare. Ad esempio riesce a classificare correttamente i protocolli peer-to-peer, e in questo senso potrebbe essere un buono strumento per la misurazione della QoS più che per il monitoraggio vero e proprio. Un altro punto a sfavore è che molti protocolli cifrati non vengono riconosciuti, ma sono classificati come "Unknown", oltre al fatto che il framework sviluppato non prevede ancora l'analisi live dei pacchetti ma solo traffico da file *pcap*.

nDPI

nDPI è una libreria scritta in C per la classificazione del traffico di rete basata su un vecchio progetto chiamato OpenDPI e attualmente mantenuta da ntop [25]. nDPI classifica il traffico analizzando in profondità tutto il payload del pacchetto. La libreria presenta un riconoscitore (dissector) per ogni protocollo, e mantiene la classificazione con la porta (o range di porte) per complementare la classificazione. Analizzando l'intero payload, si riesce a definire con la massima accuratezza il protocollo della porzione di traffico. Introduce l'estrazione degli host name server dai certificati SSL o dalle query DNS. L'accuratezza si rivela migliore e più chiara rispetto ad altre librerie in quanto nelle ultime versioni è stato suddiviso il protocollo applicativo dal protocollo vero, inoltre nDPI risulta più completa rispetto ad altre librerie perché invece di sostituire i vecchi metodi, cerca di integrarli. Lo svantaggio principale è l'utilizzo di più risorse rispetto ad altre librerie, in quanto l'unità predominante non è il singolo pacchetto, ma il flusso. Questo svantaggio viene premiato con l'estrazione di un maggior numero di informazioni e quindi avere una catalogazione migliore (ad esempio l'hash dei pacchetti BitTorrent). L'utlizzo dei flussi garantisce la possibilità di classificare molti più protocolli applicativi rispetto ad altri software.

Confrontando i vari tool tra di loro notiamo innanzitutto che le librerie open-source non si discostano moltissimo dal lavoro svolto da quelle a pagamento, col vantaggio di avere codice scritto personalmente, diffondibile e migliorabile dalla comunità, evitando di perdersi nell'oscurità dei software proprietari. Grazie ai test e ai risultati ottenuti [26], la scelta per il nostro lavoro ricade sulla libreria nDPI, sia per una maggiore completezza di informazioni, sia per alcuni lavori personalmente svolti al fine di migliorare la suddetta libreria.

2.2 Classificazioni automatica di immagini

Lo stato dell'arte per questa sezione è molto ampio. Negli ultimi anni numerose pubblicazioni riguardanti quest'ambito di ricerca sono state rila-

sciate, molte delle quali hanno dato la possibilità di notevoli sviluppi. Tra i vari lavori analizzati si è scelto quello illustrato nella pubblicazione *Applying deep learning to classify pornographic images and videos*, del Dipartimento di Informatica e Ingegneria della American University del Cairo [22]. Il lavoro svolto presenta l'implementazione di un modello di rete neurale composto da due sottoreti convoluzionali che svolgono un lavoro parallelo, i cui risultati vengono poi uniti prima di dare la classificazione dell'immagine.

La prima rete è basata sull'algoritmo *AlexNet* [23], divenuto famoso grazie alla vittoria, nel 2012, dell'*ImageNet Challenge*. Questo algoritmo è divenuto così importante che fino al 2011 i migliori classificatori avevano un errore di classificazione in media del intorno al 25%; dopo AlexNet questa percentuale è scesa al 16%. Ecco perche' esso è considerato il principale benefattore per lo sviluppo delle reti convoluzionali.
Il secondo è un modello di Google chiamato *GoogLeNet* [24] vincitore della competizione di *ImageNet* nel 2014.

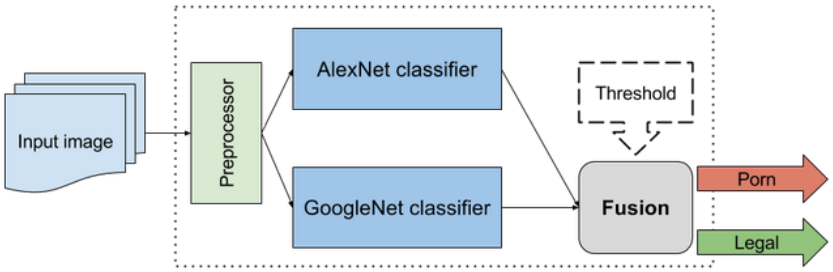

Figura 2.3: Schema del classificatore spiegato nella pubblicazione di cui sopra

Per questa rete è stata ottenuta un'accuratezza dell'87% per la prima rete (*AlexNet*), del 91% per la seconda (*GoogleNet*). Le due reti sono state utilizzate anche per la classificazione dei video, ottenendo un'accuratezza del 94%.

Nei successivi due capitoli si entrerà nel dettaglio dei lavori svolti. Prima si parlerà della libreria nDPI e dell'estensione di tale libreria; nel capitolo successivo si affronterà il problema della creazione di un personale modello di rete neurale per il riconoscimento delle immagini.

Capitolo 3

nDPI

In questo capitolo si parlerà in modo approfondito della libreria nDPI e di un riconoscitore di un nuovo protocollo ($QUIC$) che estende tale libreria.

3.1 nDPI: Open-Source High-Speed Deep Packet Inspection

Agli albori di Internet, i protocolli di traffico di rete erano identificati dall'associazione *porta/protocollo*, come già detto nel capitolo introduttivo. Per esempio SMTP utilizza il protocollo TCP sulla porta 25, mentre DNS gira su UDP sulla porta 53. Questa associazione statica tra porta e protocollo, dopo l'avvento delle *Remote Procedure Call (RPC)*, è diventata un problema. Applicazioni come *rpcbind* o *portmap* sono nate per cercare di mantenere una mappatura dinamica ma, poiché le porte maggiori della 1024 sono riservate per identificare servizi come la posta elettronica, ci sarebbe un continuo cambio di contesto tra livello utente e superutente; quelle porte sono utilizzate per servizi definiti a livello utente e generalmente dinamici.

L'identificazione dei protocolli basandosi solo sulla porta statica non è più affidabile. Un caso emblematico è l'utilizzo della porta 80 di TCP. Per default tale porta è associata al protocollo HTTP, se non fosse che l'evoluzione di HTTP deve portare necessariamente una modifica del *come* si deve riconoscere il protocollo. La diffusione preponderante di HTTP ne ha cambiato l'uso, non più limitandosi solo al traposto di pagine web, ma, per esempio, come protocollo per il download di file e trasporto di file multimediali, Questo uso pervasivo di HTTP (e successivamente HTTPS) e il suo supporto nativo sui firewall, i quali riconoscono e validano l'header dei protocolli, ha dato la possibilità a moltissime applicazioni di rete di svilupparsi al di sopra di HTTP, in modo da aggirare le politiche restrittive. Il meccanismo viene sfruttato soprattutto dai protocolli peer-to-peer, come ad esempio *Skype*, che gira sulla porta 80, fingendosi HTTP agli "occhi" dei

firewall e riuscendo a superare eventuali limitazioni di traffico. Tutto ciò ci fa riflettere su quale sia diventata la vera natura di HTTP e della porta 80.

Vien da sé che una classificazione diversa e molto più accurata è divenuta necessaria, sia per un monitoraggio completo dei servizi di rete, sia per una maggiore sicurezza vista la facilità con cui i firewall potevano essere bypassati [27].

La necessità di incrementare la visibilità del traffico classificandolo più in dettaglio, ha dato la possilità di sviluppo a molte librerie che adottano la tecnica del DPI, abbandonando o integrando il metodo delle porte. L'analisi del pacchetto comporta inevitabilmente un aumento dei costi computazionali, ma ci garantisce un netto miglioramento dell'accuratezza nella classificazione dei protocolli a livello applicativo e di quelli tradizionali.
Ognuna delle motivazioni precedenti è stata la chiave di volta per lo sviluppo di una libreria open-source per il monitoraggio di reti 10-Gbps, che analizzi il payload nella maniera più efficiente possibile senza dover ricorrere a soluzioni hardware specifiche.

3.1.1 Da OpenDPI a nDPI

La libreria OpenDPI è stata il punto di partenza per lo sviluppo di nDPI. E' scritta in C e presentava due componenti principali: il *nucleo*, responsabile della cattura del pacchetto, decodifica del livello IP e trasporto, estrazione di informazioni di base quali indirizzi IP e porte, e i *dissector*, i componenti che consentono il riconoscimento dei protocolli. nDPI ha mantenuto questo schema architetturale e lo ha esteso con altre caratteristiche per superare alcuni problemi presenti nella libreria:

- Sono state rese flessibili quelle strutture dati che OpenDPI manteneva statiche e quindi impossibili da estendere se si necessita di un nuovo protocollo o di una nuova funzionalità.
- Quando un protocollo era riconosciuto non si ritornava subito il risultato ma la ricerca andava avanti anche sugli altri dissector, degradando notevolmente le prestazioni.
- Non era presente alcun supporto per i protocolli criptati (HTTPS).
- OpenDPI non è era multi-threaded, creando problemi di concorrenza per quelle applicazioni che utilizzano più thread, costringendole ad utilizzare semafori per evitare risultati impredicibili.
- I dissector della libreria non erano basati su alcun tipo di gerarchia. Per esempio se arrivava una nuova connessione TCP sulla porta 80, OpenDPI non provava prima se quel traffico era HTTP, ma scorreva i vari dissector in ordine come registrati. Questo comporta un notevole degrado delle performance.

- La libreria non aveva alcuna possibile configurazione runtime per i protocolli. Se un utente voleva definire un protocollo per un'associazione come *UDP/port X* avrebbe dovuto scrivere un dissector specifico. Questo comporta problemi soprattutto in ambienti chiusi come le LAN, in cui i protocolli sono configurati per girare su ben precise porte. In questo caso è molto più conveniente analizzare l'header anziché tutto il payload del pacchetto.

- Non era possibile estrarre metadati, essenziali per capire i protocolli applicativi legati ad HTTP, oppure estrarre i service provider da query DNS, in modo da classificare il traffico in base a queste informazioni.

Riassumendo possiamo dire che OpenDPI è stato un ottimo punto di partenza per lo sviluppo della nuova libreria. nDPI può essere usata da qualsiasi applicazione proprio per la scelta di non utilizzare specifiche tecniche per un determinato set hardware, ma, a fronte di un maggior costo computazionale, rendere l'uso più flessibile e più scalabile. nDPI è focalizzata sulla classificazione del traffico Internet in modo da avere due livelli di riconoscimento: uno per il protocollo standard, l'altro per quello proprietario. Esclusi i protocolli conosciuti, la creazione dei dissector per quelli proprietari è stata ottenuta grazie ad un meticoloso reverse-engineering sul payload. In alcuni casi la documentazione presente in rete ha aiutato questo processo, ma per molte applicazioni (ad esempio Skype, Whattsapp) il lavoro è stato fatto senza aiuti esterni, se non limitati, il che ha reso la scrittura dei dissector una vera e propria sfida.

Inizialmente era stata introdotta la possibilità di utilizzare nDPI anche a livello kernel; in seguito si è visto che a causa del maggior utilizzo di protocolli applicativi, questo supporto risultava superfluo, soprattutto in temini di mantenimento del codice, ed è stato eliminato. Progetti paralleli come *ndpi-netfliter* [28] continuano a portare avanti questo discorso.

3.1.2 Architettura e Implementazione di nDPI

nDPI definisce un protocollo a livello applicativo con la coppia *(protocol ID, protocol name)*, ad esempio `(Whatsapp, 142)`. Questo dà la possibilità ai software che utilizzano la libreria di avere un numero univoco associato al protocollo. Il set di protocolli riconosciuti include sia quelli a livello di rete, come SMTP o DNS, che quelli applicativi. Per questo motivo i flussi di dati Facebook o Twitter sono classificati con il nome dei servizi stessi, in quanto la comunicazione avviene con server appartenenti a Facebook/Twitter: per nDPI sono entrambi protocolli.

Un protocollo è solitamente riconosciuto tramite un *dissector* scritto in C. Un dissector è un pezzo di codice con il quale si tenta di riconoscere se un pacchetto è classificabile o meno con il protocollo associato ad esso. Questo non è l'unico modo. Si può definire una classificazione anche in

termini di porta/protocollo, indirizzo IP o attributi del protocollo estratti dai metadati.

La classificazione è stata resa più chiara grazie alla suddivisione tra **application protocol** e **master protocol**. Facciamo un esempio con Skype. Skype può essere riconosciuto in due modi: tramite il proprio dissector, oppure estraendo l'host name dai metadati di HTTP o DNS. Se un pacchetto Skype viene classificato tramite il proprio dissector, si setterà solo *application protocol* come Skype, mentre se si estrae un server name tipo `.skype.com` o `.skypedata.com`, nDPI imposterà **app_proto = Skype** e **master_proto = DNS**, nel caso l'host name sia estratto da una query DNS: il flusso di dati che ne consegue sarà considerato Skype.

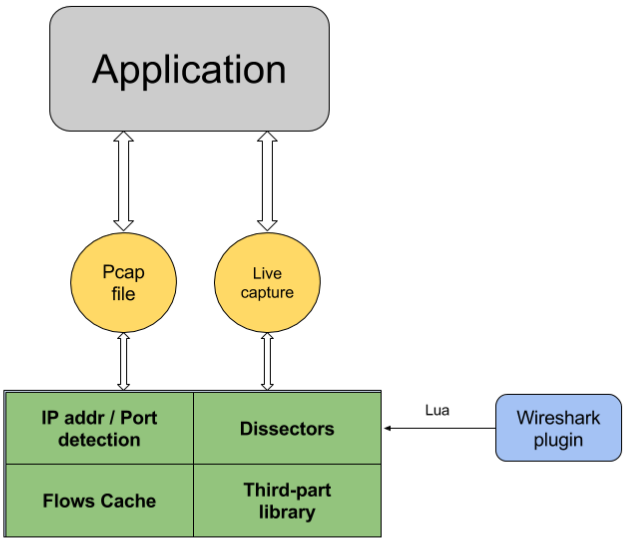

Figura 3.1: Architettura della libreria nDPI

La libreria eredita l'architettura del progetto OpenDPI, con il codice di base per implementare le funzioni generali, e i dissector per il riconoscimento dei protocolli. Successivamente il codice è stato reso *rientrante*, in modo tale che la libreria sia *thread-safe* ed evitare l'uso di lock o altre tecniche per la serializzazione da parte dell'applicazione chiamante. Questo perché OpenDPI non era multithreading, abbassando le performance e alzando i tempi di riconoscimento e classificazione dei flussi di pacchetti.

La struttura dati su cui nDPI basa il proprio lavoro è il **FLUSSO**,

ovvero una 5-tupla composta da ⟨ Source IP address, Destination IP address, Source Port, Destination Port, Transport Protocol ID ⟩. I flussi sono importanti poiché nDPI non si prefigge di analizzare e classificare i protocolli in base solamente alle informazioni dei pacchetti, bensì vuol andare oltre e poter conservare dati aggiuntivi. Inoltre ogni flusso mantiene lo stato dei dissector che sono stati scartati, grazie ad una maschera di bit rappresentante tutti i protocolli implementati. Un flusso è composto da due strutture C distinte, una per i flussi *tcp* e una per quelli *udp*, raccolte all'interno di un'unica struttura così descritta

```
struct ndpi_flow_struct {
  ...
  ...
  union {
    struct ndpi_flow_tcp_struct tcp;
    struct ndpi_flow_udp_struct udp;
  } l4;
  ...
  struct ndpi_id_struct *server_id;
  u_char host_server_name[256];
  u_char detected_os[32];
  u_char bittorent_hash[20];
  ...
  struct {
    ndpi_http_method method;
    char *url, *content_type;
  } http;
  ...
  struct {
    u_int8_t num_queries, num_answers, reply_code;
    u_int16_t query_type, query_class, rsp_type;
  } dns;
  ...
  struct {
    char client_certificate[48], server_certificate[48];
  } ssl;
  ...
}
```

Il codice illustra alcuni dei principali campi contenuti all'interno del flusso; ognuno di essi è importante per i discorsi fatti precedentemente riguardo la qualità e quantità dei protocolli che possiamo riconoscere.

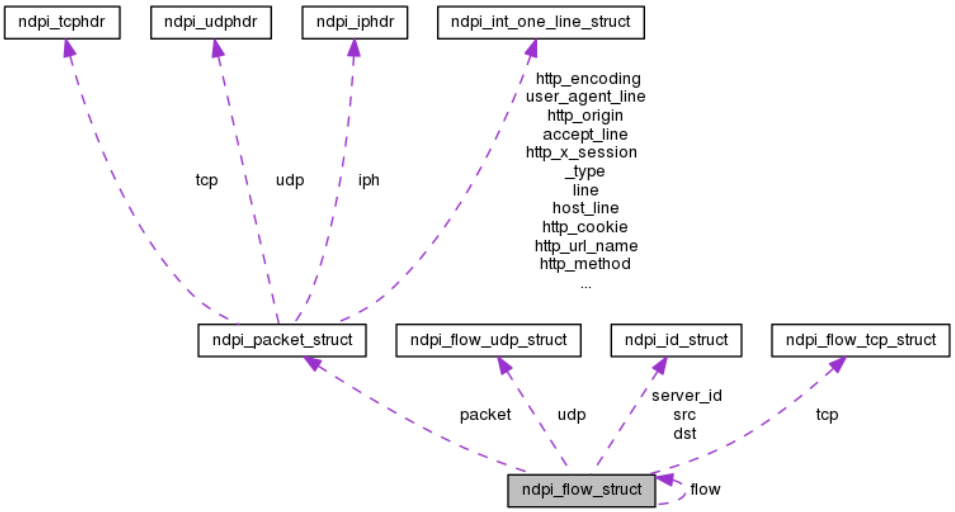

Figura 3.2: Diagramma di collaborazione per ndpi_flow_struct

Da questo diagramma possiamo notare come, all'interno del flusso, sia contenuta anche la struttura rappresentante il singolo pacchetto da analizzare (`ndpi_packet_struct`), ovvero ciò che viene passato ai dissector per essere analizzato; ogni informazione sul singolo pacchetto contribuisce alla caratterizzazione del flusso e, quindi, del traffico.

L'inizializzazione della libreria avviene durante la fase di start-up, in modo da non avere penalizzazioni nel momento in cui arriva un nuovo pacchetto. Quando l'applicazione chiamante passa il pacchetto a nDPI, la libreria procede con la decodifica del pacchetto dal livello di Rete in poi, essendo interessati agli indirizzi IP, porte e header applicativo. I dissector dei protocolli sono registrati con alcuni attributi di default. Si può specificare su quale porta TCP o UDP il protocollo gira, in modo tale da velocizzare il processo di scansione, evitando di analizzare un dissector per un protocollo TCP se la porta specificata è UDP, o viceversa. Ad esempio il dissector di HTTP specifica la porta di default TCP 80, quello di DNS TCP/UDP 53. Questo permette anche di dare una priorità ai protocolli, in quanto se dall'applicazione arriva un pacchetto con la porta 80, nDPI per prima cosa controlla se tale pacchetto sia HTTP. Se l'esito è negativo, il controllo è passato ai rimanenti dissector. In questo modo si riduce notevolmente il numero di passaggi che si devono effettuare. Un altro fattore cui tenere conto è il **guessing**: quando un protocollo non è ancora stato classificato, nDPI può comunque dare un'etichetta basandosi sul livello di trasporto del pacchetto. Un flusso può non essere stato classificato non solo perché non è riconosciuto, ma anche se non tutti i flussi sono analizzati dal principio. Un

caso tipico è quando un protocollo, per essere classificato, ha bisogno di `n` flussi, ma l'applicazione ne ha mandati al massimo `n-1`.

Il ciclo di vita del processo di classificazione può essere schematizzato come segue:

- Quando arriva un pacchetto dall'applicazione, si valuta se abbia l'header di Rete o meno (i pacchetti ARP, ad esempio, vengono scartati)
- Successivamente il pacchetto viene passato alla funzione di decodifica di Rete e Trasporto, in cui si estraggono le informazioni sulle porte (TCP o UDP) e sugli indirizzi IP che verranno passati alla funzione principale del nucleo della libreria
- Con le informazioni estratte nel passaggio precedente si invoca la funziona vera di decodifica, e si cerca di stabilire se il pacchetto arrivato appartiene ad un flusso esistente oppure è uno nuovo. Nel primo caso aggiorniamo eventuali campi con le nuove informazioni; nel secondo inizializiamo un nuovo flusso
- Arrivati all'analisi del payload, la funziona cerca una possibile corrispondenza con uno dei dissector disponibili: se la trova, si setta il campo *app_proto* con l'ID per il protocollo associato, altrimenti passa il controllo al successivo dissector. Se non avviene alcun match ci sono due opzioni: o il dissector viene settato nella maschera dei protocolli esclusi, oppure la classificazione viene lasciata al pacchetto successivo del flusso. Nel caso in cui anche questa opzione fallisca, il pacchetto verrà etichettato come **Unknown** e si tenterà la classificazione con la tecnica del guessing, quindi valutando se porte o indirizzi noti.

Una domanda tipica che sorge è *quanti pacchetti servono alla libreria per poter classificare il traffico ?* La risposta è abbastanza ovvia: dipende dal protocollo. In generale si è visto però che la classificazione dei protocolli basati su UDP richieda meno pacchetti in generale rispetto a quelli TCP, anche se vi sono delle eccezioni, come ad esempio BitTorrent.

3.1.3 Traffico criptato

La vera sfida per le librerie basate sul DPI è la classificazione del traffico cifrato. Il sempre maggior numero di protocolli proprietari uniti all'incremento della volontà di sicurezza (e anonimato) ha reso necessario un diverso approccio al problema. E' impossibile scrivere dissector per molte applicazioni di questo genere, salvo estrarre metadati direttamente dal livello SSL/TLS che ci permettano di avere informazioni sufficienti per capire la

natura del flusso. L'unica parte dei dati scambiati che è possibile decodificare in una sessione SSL avviene durante la fase di handshaking tra il server e il client. Grazie al dissector specifico per i protocolli `SSL 1.0/2.0/3.0` e `TLS 1.0/1.1/1.2` (l'ultima versione 1.3 è ancora in fase di sperimentazione), si estrae il server name dal certificato del client (precisamente dal pacchetto `Client Hello`). L'informzone è salvata nel flusso corrispondente con la stessa modalità di estrazione dei metadati di HTTP.

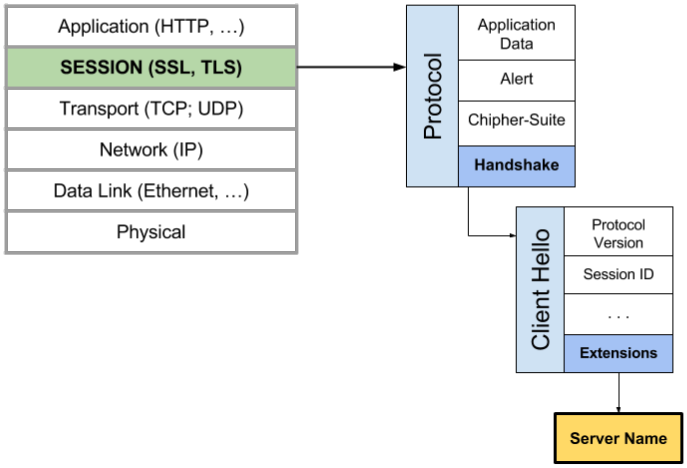

Figura 3.3: Estrazione del Server name in nDPI

Con questo approccio si riesce ad identificare le applicazioni che sarebbero impossibili da classificare normalmente. Ad esempio comunicazioni cifrate il cui server name è `.twttr.com`, `.fbcdn.net` o `nflxvideo.net` sono classificate rispettivamente come `Twitter`, `Facebook` e `Netflix`. nDPI tuttavia specifica che questi protocolli derivano da traffico criptato settando il master protocol con l'ID di SSL/TLS. L'estrazione dei certificati consente alla libreria di porre un tassello aggiuntivo in termini di sicurezza, in quanto è possibile scoprire se ci sono dei cosiddetti *Self-signed certificate*, ovvero certificati d'identità firmati dalla stessa entità che li ha generati, anziché da un ente di terze parti.

Un ulteriore metodo di classificazione consiste nella possibilità di creare un file di configurazione a runtime da parte degli utenti, in modo da definire nuovi protocolli per nome. Quando nDPI scopre che un nuovo nome è

già definito, il file non fa altro che estendere la configurazione già presente in nDPI, classificando il protocollo associato con il nuovo nome definito dall'utente.

Ad esempio è possibile creare il file `protos.txt` con le seguenti personalizzazioni:

```
tcp:3000@ntop
host:"inter.it"@Triplete
ip:213.75.170.11@PersonalProtocol
```

Vediamo ora un altro modo con cui nDPI può riconoscere i protocolli.

3.1.4 Algoritmo Aho-Corasick per il riconscimento delle stringhe

Come già spiegato in precedenza, la libreria non utilizza solo dissector specifici, ma valuta anche l'associazione con porte, indirizzi IP e host name definiti come stringhe. Soffermiamoci sul modo in cui vengono trattati questi ultimi.

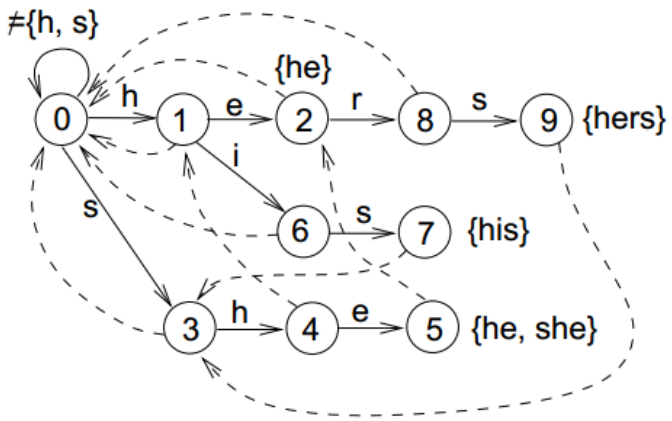

Figura 3.4: Implementazione di un Automa a Stati Finiti per l'algoritmo Aho-Corasick

Le stringhe estratte dai metadati sono salvate in un automa basato sulla libreria Multifast [29], un software open-source che permette la ricerca e la manipolazione di stringhe in modo efficiente e veloce *al volo*. La forza di questa libreria è la possibilità di trattare molte più stringhe in un unico passaggio, fornendo API flessibili per garantire al programmatore di determinare esattamente quale sottoinsieme di stringhe occorrono per avere un match,

e interrompere la ricerca una volta raggiunto l'obiettivo. Si possono definire dei pattern per il confronto tra le stringhe e i campi prestabiliti dall'utente. All'avvio, la libreria crea l'automa che gestirà ogni possibile contronto una volta che nDPI recupererà una stringa dai metadati del pacchetto.

L'algoritmo Aho-Corasick è stato sviluppato appositamente per la ricerca veloce di pattern all'interno delle stringhe e quindi poter fare confronti veloci ed efficienti. L'algoritmo costruisce un automa a stati finiti simile ad un *trie* [30] con link addizionali tra i vari nodi interni. Grazie a questi ulteriori collegamenti è garantita una ricerca dei pattern molto più veloce rispetto ad un normale *prefix tree*.

3.1.5 Wireshark plugin

Una delle ultime funzionalità aggiunte alla libreria è il supporto per Wireshark. Wireshark permette ad ogni utente di implementare dissector scritti in **lua**, così da estendere il pacchetto dei protocolli riconosciuti. Partendo da questo punto, nDPI è stato esteso in modo da complementare la codifica interna dei protocolli di Wireshark. Per fornire un supporto a questa implementazione, si utilizza la demo *ndpiReader*, un'applicazione creata per verificare la correttezza del lavoro generale svolto da nDPI. ndpiReader fornisce l'analisi del protocollo nDPI, e un plugin di Wireshark interpreta le informazioni della libreria.

Durante la cattura, il plugin di ndpiReader passerà a Wireshark le informazioni del protocollo aggiungendo un pacchetto ethernet in cui sono presenti i dati del protocollo nDPI. In questo modo è possibile anche definire ulteriori filtri per Wireshark usando il meccanismo predefinito di filtraggio. Ad esempio `ndpi.protocol.name==BitTorrent` per filtrare tutto il traffico BitTorrent.

3.2 Implementazione del dissector di un nuovo protocollo per nDPI

Illustriamo ora uno dei principali lavori effettuati su nDPI basato sul Deep Packet Inspection. Verrà mostrata l'implementazione di un dissector per un nuovo protocollo, e le modifiche effettuate per classificare i flussi applicativi

3.2.1 QUIC: Quick UDP Internet Connections

QUIC è un protocollo sviluppato da Google nel 2013 dopo l'avvento del cosiddetto HTTP 1.2.
HTTP è il protocollo più usato a livello applicativo, ma negli ultimi anni si è visto anche come esso rappresenti un possibile collo di bottiglia. La diffusione di molte applicazioni sopra HTTP ha evidenziato un'inefficienza nel trasposto delle informazioni, causato soprattutto dal dover stabilire connessioni affidabili, a livello di trasporto, tra *client* e *server*. Un primo passo è stato fatto con lo sviluppo del protocollo *SPDY* [31], il quale introduce funzionalità importanti:

- molteplici richieste concorrenti HTTP collassate in una singola socket TCP
- compressione degli header HTTP
- client-server *push-based*; se il server è a conoscenza di risorse necessarie al client, le invia appena possibile.

Il problema di SPDY riguarda soprattutto il primo punto. Abilitare più connessioni su una sola web-socket diminiuisce il numero di porte da gestire, ma la perdita di un solo pacchetto congestiona le N connessioni su quella socket. L'altro grande svantaggio è rappresentato dalla latenza iniziale della connessione: esso dipende in ogni caso dall'handshake TCP (1 RTT) o SSL/TLS (3 RTT). Notiamo quindi che l'inefficienza è dovuta all'uso di TCP a livello trasporto.

Il protocollo QUIC risolve molte delle problematiche evidenziate sopra. L'idea principale dietro QUIC è di utilizzare UDP per risolvere i problemi che SPDY aveva a livello trasporto, includendo il tutto in un protocollo criptato per la sicurezza dei dati chiamato "QUIC Crypto", il quale provvede al trasporto sicuro in stile TSL. Siccome UDP non è affidabile e non ha il controllo della congestione, l'affidabilità è gestita a livello applicativo.

Multiplexing QUIC processa più flussi sulla stessa connessione UDP. Nella figura 4.5 vediamo come un client HTTP possa processare solo una risorsa alla volta, a differenza di QUIC (figura 4.6) dove un client riceve

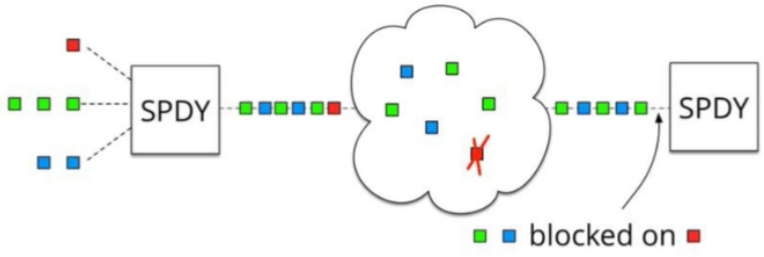

Figura 3.5: Rallentamento di SPDY causato dalla perdita o dall'arrivo fuori ordine di un pacchetto

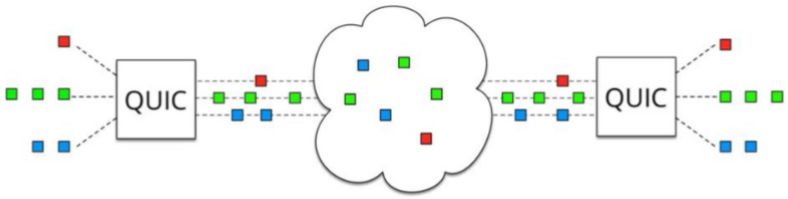

Figura 3.6: Scenario di QUIC che risolve il problema del multiplexing di SPDY

e richiede più connessioni HTTP sulla stessa socket UDP. In questo modo si elimina il principale problema di SPDY, ovvero il *head of line blocking (HOL)* causato dall'apertura di più connessioni HTTP.

Latenza e Sicurezza Un altro problema riguarda la latenza. Come vediamo dalla figura 4.7 una connessione TCP richiede un Round Trip Time per la fase di handshake nello stabilire ogni volta una connessione, che diventano ben tre se la connessione è cifrata con SSL/TLS. Quando è utilizzato QUIC si paga solamente il primo RTT, il quale diventa zero nel caso il client si sia già connesso al server in precedenza. La latenza è nulla anche nel caso di connessioni cifrate, in quanto QUIC provvede già ad una cifratura grazie ad un suo algoritmo: la decodifica avviene in modo indipendente trai vari pacchetti, evitando la perdita di tempo nella serializzazione dei pacchetti.

Connection IDentifier (CID) Una connessione QUIC è unicamente identificata da un univoco identificatore chiamato CID (Connection Identifier) a livello applicativo. Un primo vantagio rispetto alla classica identificazione basata sugli indirizzi IP e numeri di porta, è che ogni trasferimento tra due reti può essere trasmesso senza bisogno di ristabilire una connessione. Inoltre l'utilizzo del CID evita i problemi derivanti dal rinezionamento di una connessione sotto NAT, poiché la conessione QUIC è indipendente

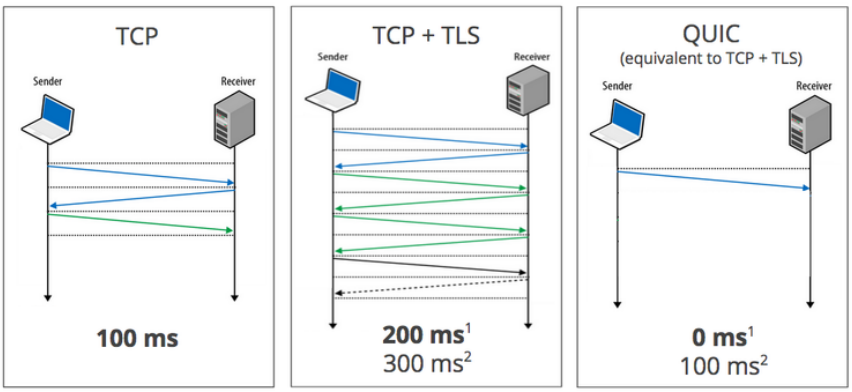

Figura 3.7: Comparazione tra TCP, TCP + TLS e QUIC [32]

dai nuovi indirizzi IP. Infine l'utilizzo di CID permette un supporto nativo per il cosiddetto *multi-path*: un dispositivio può usare tutte le connessioni disponibili con un unico CID.

Come riportato in alcuni test di Google [33] *These benefits are even more apparent for video services like YouTube. Users report 30% fewer rebuffers when watching videos over QUIC.*

Dissector per QUIC

Lo studio approfondito del protocollo ha permesso d implementare il dissector di QUIC in nDPI.

Per prima cosa, ogni dissector in nDPI ha una funzione per inizializare il dissector. In questo caso:

```
void init_quic_dissector(ndpi_detection_module_struct *ndpi_struct,
                         u_int32_t *id,
                         NDPI_PROTOCOL_BITMASK *detection_bitmask)
{
  ndpi_set_bitmask_protocol_detection("QUIC", ndpi_struct,
             detection_bitmask, *id,
             NDPI_PROTOCOL_QUIC, ndpi_search_quic,
             NDPI_SELECTION_BITMASK_PROTOCOL_V4_V6_UDP_WITH_PAYLOAD,
             SAVE_DETECTION_BITMASK_AS_UNKNOWN,
             ADD_TO_DETECTION_BITMASK);

  *id += 1;
}
```

Una cosa interessante da notare è la macro
`NDPI_SELECTION_BITMASK_PROTOCOL_V4_V6_UDP_WITH_PAYLOAD`
dove si specifica il tipo di indirizzi IP che si possono trovae e se abbiamo a

che fare con UDP o TCP a livello trasporto. Gli altri due campi `SAVE_DETECTION_BITMASK_AS_UNKNOWN` e `ADD_TO_DETECTION_BITMASK` rappresentano rispettivamente le maschere di bit per i protocolli esclusi e per quelli riconosciuti.
Analiziamo ora la funzione `ndpi_search_quic`:

```
void ndpi_search_quic(ndpi_detection_module_struct *ndpi_struct,
                      ndpi_flow_struct *flow)
```

All'arrivo di un nuovo pacchetto, la libreria lo mantiene dal livello trasporto in poi, nella struttura `ndpi_struct`, riportando anche il flusso corrispondente.

Per prima cosa si cerca di capire se il pacchetto è UDP o TCP; nel secondo caso si scarta, in quanto QUIC gira su UDP. Da qui inizia la parte di dissecting specifico per il protocollo, analizzando il frame e cercando una possibile corrispondenza tra i byte del pacchetto e le caratteristiche del specifiche di QUIC.

Come mostra la figura 4.8 i campi sempre presenti di un header QUIC sono `Public Flag`, `CID`, `Version`, `Sequence Number`.

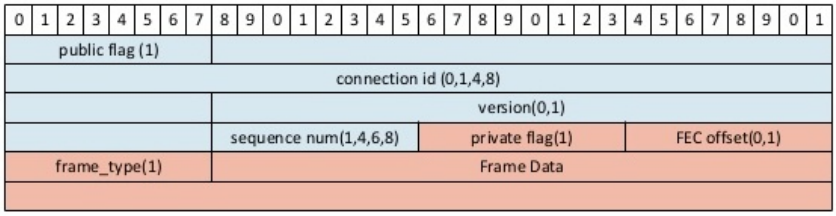

Figura 3.8: Formato di un Frame QUIC

Siccome QUIC sostituisce TLS, il primo pacchetto di una connessione sarà di tipo Client Hello (proprio come SSL/TLS), e rappresenterà un nuovo flusso. Tale pacchetto ha più campi rispetto ad un pacchetto dati, il che comporta molte più informazioni, come ad esempio lunghezza del payload, lo User Agent utilizzato, e soprattutto il *server name*. Proprio grazie all'estrazione di questo campo possiamo dare una classificazione più precisa del flusso, verificando a quale protocollo corrisponde la stringa nel file chiamato `ndpi_content_match.c.inc`, in cui sono presenti indirizzi IP e nomi dei server noti. Il campo dedicato al server name nell'header QUIC prende il nome di *Server Name Indication (SNI)*.

Il seguente codice mostra la chiamata alla funzione per cercare il protocollo applicativo corrispondente al nome del server estratto.

```
ndpi_match_host_subprotocol(ndpi_struct,
                    flow,
                    flow->host_server_name,
```

```
                    strlen((const char*)flow->host_server_name),
                    NDPI_PROTOCOL_QUIC);
```

Lo sviluppo del protocollo QUIC ha rivoluzionato il modo di trasferimento dei file e l'apertura delle connessioni tra host: se prima, in un sessione di cattura del traffico si vedeva solo traffico SSL, adesso si nota come alcune applicazioni, soprattutto multimediali**(YouTube)**, facciano un uso massiccio di QUIC, in modo da velocizzare soprattutto il buffering dei video.

La scrittura del dissector di questo protocollo è stata molto importante per dare un incremento alla classificazione del traffico in nDPI, in quanto ancora nessun software open-source di DPI aveva scritto un dissector esaustivo per questo protocollo.

Fino ad ora abbiamo affrontato l'argomento della classificazione basandoci sulla naturale evoluzione delle tecniche di ispezione, dalle porte, passando per l'analisi degli header, fino ad arrivare al più avanzato Deep Packet Inspection. Come abbiamo visto il DPI ci consente di analizzare con precisione assoluta i byte del pacchetto in modo tale da estrarre pattern e confrontarli con le specifiche dei protocolli o, nel caso di applicazioni cifrate, classificare i flussi di pacchetti in base al server name estratto dal pacchetto Client Hello di TLS/SSL o di QUIC, oppure da una query DNS. Il problema è che il cambiamento repentino di molti protocolli, insieme alla maggiore complessità della classificazione delle applicazioni a livello di rete, non permette più alla tecnica del DPI di essere esaustiva. A questo punto è stato naturale chiedersi se si può pensare ad una via parallela.

Una possibile risposta ci viene data dalle richieste in ambito monitoraggio di rete. Negli ultimi tempi la questione è rappresentata dalla possibilità di avere o meno, per le proprie intranet (specie se di grandi dimensioni), un modo per dare una caratterizzazione dei contenuti del traffico viaggiante in rete, quindi immagini, testo e file multimediali.
Proprio grazie alla volontà di porre un ulteriore livello di caratterizzazione, è nata l'idea di costruire un riconoscitore per la classificazione automatica del contenuto dei pacchetti, in particolare per le *immagini*, in modo da inserirlo in un prototipo di framework che utilizzi Deep Packet Inspection e Machine Learning.

Nel successivo capitolo parleremo di Machine Learning, in particolare del Deep Learning.

Capitolo 4

Machine Learning

4.1 Introduzione al Machine Learning

La risoluzione di un problema informatico necessita di un algoritmo. Ad esempio, per ordinare una sequenza di numeri, adottiamo un algortimo di *sorting*: l'input è un insieme di numeri, l'output è lo stesso insieme di numeri ordinati. Per la risoluzione del problema ci sono molte strade, ognuna con una propria efficienza in base alle diverse necessità e casi d'uso.
Per alcuni problemi non è possibile avere alcun algoritmo di questo tipo. Prendiamo il caso della classificazione tra email accettabili e spam: conosciamo l'input (*email*), conosciamo l'output (*Spam/Non Spam*), non sappiamo però come poter generalizzare la trasformazione dell'input in output se non analizzando le singole mail una ad una.

Quello che ci manca nella conoscenza, lo compensiamo con i dati [34]. In pratica, rifacendoci all'esempio precedente, vogliamo che il nostro computer (Machine) estragga il miglior algoritmo possibile dai dati passati, in modo da imparare (Learning) automaticamente quali mail siano spam e quali no. Il problema principale è costituito dalla creazione di un buon dataset, poiché più i dati in nostro possesso saranno quantitativamente e qualitativamente alti, maggiore sarà l'accuratezza ottenibile. E' impossibile pensare di costruire un modello perfetto; è fattibile costruire una buona e utile approssimazione della realtà. Le tecniche di ML si basano sul concetto di estrazione di *pattern* comuni tra i dati, in modo da categorizzare questi in base alle esigenze dell'utente. Pattern usati non solo per imparare, ma anche per effettuare predizioni.

Il Machine Learning, oltre a toccare problemi di database (*data mining*), è parte di Intelligenza Artificiale. Per essere intelligente, un sistema che si trova in un ambiente dinamico, deve essere in grado di imparare.

Riassumendo il concetto, con il Machine Learning si vuole rappresentare al meglio uno scenario reale costrunedo un modello, basandosi sui dati raccolti per quell'ambiente. Il ML usa la statistica per la creazione dei modelli

matematici, poiché il processo di creazione è di deduzione (o inferenza) da esempi concreti.

Per lo sviluppo di qualsiasi modello, vi è una fase di *training* imprescindibile. Successivamente, una volta che il modello ha imparato, i risultati devono essere validate con un'alta efficienza nella fase di *testing*.

4.1.1 Concetti chiave del Machine Learning

Association rule Per capire questo concetto facciamo un esemoio: se una persona che acquista un prodotto X di solito compra ache un prodotto Y, ed esiste un'altra persona che compra un prodotto X ma non quello Y, quest'ultima è un potenziale acquirente di X. Cercando una *regola associativa* si cerca di imparare una probabilità condizionata del tipo $P(Y|X) = \frac{P(Y \cap X)}{P(X)}$, dove Yè il prodotto da condizionare con X (il prodotto già acquistato dal cliente). Per rendere la ricerca ancora più precisa per un certa categoria, si aggiunge anche il set di attributi D per tale categoria, ottenendo $P(Y|X, D)$.

Classification Lo scopo di tutto è riuscire a dare una classificazione in base a categorie prestabilite. Dopo la fase di *training* dei dati, una regola di classificazione può essere formalizzata nella seguente condizione:

IF E1 $>$ θ THEN C1 ELSE C2

Se un certo evento supera una certa soglia θ allora avremmo una classificazione C1, altrimenti C2 (con C1 e C2 categorie stabilite a priori).

Prediction Avendo una regola come quella sopra, l'applicazione principale è la *predizione*. Essa rappresenta la percentuale con cui un dato è classificato. Quindi possiamo dire che la classificazione è determinare se un dato è X o Y, mentre la predizione rappresenta con quale probabilità riusciamo ad identificare correttamente un dato X o Y.

Regression Per capire che si intende per *regressione* facciamo un altro esempio concreto. Abbiamo un sistema che deve predire il prezzo di un auto in base agli attributi di tale auto (marca, anno, cilindrata, cavalli motore, etc.). L'output è il prezzo. Il problema di tirare fuori il costo è un problema di regressione.
Se X denota gli attributi dell'auto e Y il prezzo, possiamo istruire un modello in cui il programma utilizza una funzione per i dati in modo da imparare Y in funzione di X. In termini matematici:

$$y = Wx + W_0 \tag{4.1}$$

per valori ideali di W e W_0.

I problemi con responso *quantitativo* sono quelli di ***regressione***; quelli *qualitativi* invece sono problemi di ***classificazione***.

Unsupervised Learning In un approccio **non supervisionato** non c'è alcuna possibilità di controllare i risultati in base ad un output prestabilito; l'unica cosa in nostro possesso sono i dati di input. Il lavoro è basato sulla ricerca di pattern occorrenti in modo da generalizzarli e farli diventare regole di classificazione o regressione. Ciò è chiamato *densità di stima*.
Uno dei metodi principali utilizzati è il *clustering*: si dividono gli input in gruppi (cluster) e si cercano dei pattern comuni per ottenere il miglior risultato possibile. Un esempio tipico è la predizione delle migliori mosse in una partita di scacchi. Nessuno può sapere quale sia l'output migliore ma, in base alle mosse in input, è possibile trovare la scelta più vantaggiosa per vincere la partita.

L'altro approccio opposto è quello **supervisionato**, in cui il lavoro consiste nella ricerca della migliore mappatura tra input e output, e il confronto dei risultati con quelli reali, così da poter aggiustare i parametri del modello della fase di training nel caso ci sia un alto errore.

4.1.2 Supervised Learning

Approfondiamo il discorso sul *supervised learning* poiché ci servirà per capire meglio la teoria dietro il modello sviluppato per la caratterizzazione del contenuto dei pacchetti.

Quando decidiamo quali siano le classi di rappresentazione di un modello (il nostro output), è fondamentale scegliere le caratteristiche importanti che ci garantiscano la scelta corretta di una classe rispetto ad un'altra, o, nel caso della regressione, quale sia la percentuale corretta più alta che si avvicina alla realtà. Questo approccio, detto *input representation*, dà importanza a certi attributi e ne ignora altri.
Lo scopo è quello di adattare il modello per far relazionare le risposte con le predizioni, puntando ad avere risposte accurate per future osservazioni (*predizione*) o per migliorare la relazione tra le risposte ottenute (*inferenza*). Per fare un esempio legato al nostro caso, si deve determinare quali siano le principali caratteristiche per riconoscere se un'immagine è illecita o meno.

Denotiamo con x_1 il primo attributo, con x_2 il secondo, con x_t il t-esimo attributo. Questo rappresenta l'insieme degli attributi per definire il nostro discriminante:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_t \end{bmatrix}$$

e con r le etichette con cui classifichiamo i dati di input:

$$r = \begin{cases} 1, & \text{se } x \text{ è Lecita} \\ 0, & \text{se } x \text{ è Illecita} \end{cases}$$

Ogni immagine è quindi rappresentata come la coppia (x, r) e il training set contiene N esempi del tipo

$$X = \{x^t, r^t\}_{t=1}^N \tag{4.2}$$

L'algoritmo di *learning* cerca l'ipotesi particolare $h \in H$ per approssimare al meglio la predizione che si avvicina di più ai valori reali in X.

Lo scopo quindi è la ricerca della migliore ipotesi $h \in H$ per meglio rappresentare ciò che il modello impara nella fase di training, ovvero

$$r = \begin{cases} 1, & \text{se } h \text{ classifica x come Lecita} \\ 0, & \text{se } h \text{ classifica x come Illecita} \end{cases}$$

Misurare la qualità del modello Per valutare le performance di un modello di apprendimento su un dataset, si utilizzano tre valori di misurazione:

- **Precision** è il rapporto tra $\frac{tp}{tp+fp}$, dove tp è il numero dei veri positivi (*true positive*), fp quello dei falsi positivi (*false positive*). La *precision* è l'abilità del modello di non classificare come positivi i dati che dovrebbero essere etichettati come negativi, in rapporto ad una categoria.
- **Recall** è il rapporto tra $\frac{tp}{tp+fn}$, dove tp è il numero dei veri positivi (*true positive*), fn quello dei falsi negativi (*false negative*). La *recall* è l'abilità del modello di classificare tutti i dati di una classe correttamente.
- **F1-Score** è la misura di quanto *precision* e *recall* siano in armonia tra di loro. Più il valore si avvicina ad 1, migliore è il valore ottenuto sia dalla *precision* che dalla *recall*.

Overfitting Avere un basso MSE sul trainig set non necessariamente è sintomo che il nostro modello è stato costruito in modo corretto. Un problema più grave dell'underfitting è il cosiddetto *overfitting*.
Questo accade perché la procedura del nostro modello statistico sta "lavorando troppo" nel cercare pattern all'interno dei dati. Esagerando con l'apprendimento, la funzione mantiene pattern che sono dati da scelte casuali e che non trovano riscontro poi nella successiva fase di *prediction* sul test set. Possiamo dire che il modello ha imparato troppo bene e che non

riesce a riconoscere nulla al di fuori del training set. E' come se avessimo un test set enorme rispetto al training, e i supposti pattern, trovati all'interno del training set, non fossero riscontrati nel test set perché non esistono.

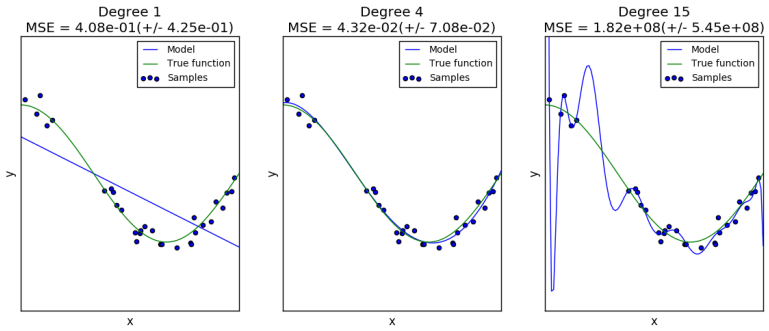

Figura 4.1: Grafici rappresentanti Underfitting (sinistra), Overfitting (destra) e approssimazione corretta (centro)

Il nodo da sciogliere è quello per cui si deve trovare il minimo punto per cui la funzione approssima al meglio la curva reale, in modo da fermare al momento giusto la fase di training per evitare l'overfitting. Uno dei metodi più usati è il *cross-validation*, che sarà spiegato in seguito, il quale cerca di stimare il MSE del test usando il training set.

4.1.3 Classificazione

Come detto precedentemente, i problemi di regressione assumono che la risposta ottenuta dal modello sia di tipo *quantitativo*. Per fare questo i modelli adottano un approccio chiamato *linear regression*, col quale si cerca di approssimare al meglio i risultati utlizzando funzioni lineari.
In altre situazioni invece abbiamo bisogno di una risposta *qualitativa* da parte del modello. Lo studio che sta alla base si chiama ***classificazione***.

Predire una risposta qualitativa per un'osservazione significa classificare una data risposta, poiché non si fa altro che etichettare il risultato con una classe o una categoria prestabilita.

Volendo ottenere un risultato diverso dai modelli di regressione, si necessita di una tecnica diversa, chiamata *logistic regression*.

Come avviene la classificazione ?

$$Y = Wx_i + b \tag{4.3}$$

Tale equazione rappresenta la *linear classification*, dove W è la *matrice dei pesi* associata all'immagine, di dimensione K x D, (K numero delle categorie e D dimensione dell'immagine), b è il vettore "bias" (*bias vector*)

di dimensione K x 1, x_i è la *i-esima* immagine e Y è l'output (la categoria associata). Da notare che il vettore b influenza il risultato dell'output senza però interagire con i dati di input.

Il vantaggio di questo approccio, che sta alla base del metodo di classificazione, è quello di utilizzare i dati del training per imparare i valori di W e b ma, una volta finito il processo di *learning*, l'intero training set è scartato per non influenzare la predizione che si effettuerà sul test set, così da poter **validare il modello**.
Il risultato ottenuto dall'equazione è associato ad ogni categoria.

$$Y = \begin{cases} 2.0 \\ 1.0 \\ 0.1 \end{cases}$$

In questo esempio vediamo come la prima categoria sia quella che ha ottenuto uno *score* maggiore (2.0). I risultati ottenuti rappresentano la percentuale che il dato in ingresso sia di una categoria a rispetto ad una categoria b.

Il passo successivo è la trasformazione degli *score* in *probabilità* normalizzate. Questo processo avviene grazie ad una funzione nota con il nome di **softmax**, la quale computa ognuno dei risultati associati ad una categoria in probabilità. La funzione softmax, chiamata anche *funzione di normalizzazione esponenziale*, "appiattisce" un vettore y di dimensione K in un vettore $\sigma(y_j)$ (K numero delle classi). Il *Softmax layer* è utilizzato come livello finale per ottenere un output con valori tra 0 e 1.

$$\sigma(y_j) = \begin{cases} 0.7 \\ 0.2 \\ 0.1 \end{cases}$$

Se non vogliamo avere un modello di regressione che ci dià le percentuali delle varie categorie, si deve poter avere come risultato un solo valore rappresentante la classe predetta. Per fare ciò si trasformano gli *scores* in un vettore chiamato *one hot encoding*, il quale ha come risultati possibili 0 e un solo 1, la nostra classe predetta.

Questo ci permette di avere avere

$$D = \begin{cases} 1.0 \\ 0.0 \\ 0.0 \end{cases}$$

da cui prendiamo il valore massimo $argmax(x)$, in questo caso 1.0, la prima categoria.

I risultati ottenuti ci soddisfano ? Come possiamo migliorare eventualmente i parametri per cui il modello generi il minor errore possibile (*loss*) ed ottenere una miglior accuratezza ?

Loss La funzione *loss* è molto importante perché misura la qualità del set di parametri utilizzati nella fase di training. Il valore che otteniamo sarà tanto più basso quanto migliore è il modello istruito: l'obiettivo sarà quello di rendere minimo il valore restituito da questa funzione. In sintesi, la *loss function* ci quantifica quanto sia "buona" o "cattiva" una funzione di classificazione (nel nostro caso la *softmax*).

La funzione di loss in un classificatore che utlizza la funzione softmax, è chiamato *cross-entropy loss*.

$$L_i = -\log(\frac{e^{s_{y_i}}}{\sum_j e^s_j}) \tag{4.4}$$

dove s è il risultato ottenuto dalla funzione di classificazione, mentre la funzione di softmax è moltiplicata per il logaritmo naturale negativo per ottenere il valore dell'errore sul risultato della classificazione. La funzione è definita su un dato *i-esimo*. Il valore di loss deve essere computato sull'intero dataset, quindi la funzione completa diventa la seguente

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i \tag{4.5}$$

Optimization Abbiamo visto che un set di parametri che predicono correttamente un dato x_i ha un basso valore di *loss*. Come possiamo garantire ed essere sicuri che questo valore sia il minimo possibile nel nostro scenario ? La risposta è **ottimizzazione**.

L'ottimizzazione è il processo per trovare il miglior set di parametri W che minimizzano il la funzione di loss. Per fare questo vi sono diverse tecninche applicabili. Una delle più quotate, soprattutto per il Deep Learning, e che utilizzeremo anche per il modello convoluzionale, è la *computazione del gradiente*.

Partendo da un valore di loss x_0, si deve poter calcolare il passo successivo per poterci muovere in direzione del valore minimo oltre cui il modello non è più ottimizzabile.

La direzione in cui i valori si muovono è relazionata col gradiente della funzione di loss. Matematicamente lo *step* è definito come

$$\alpha = \delta L(W_i, W_{i+1}) \tag{4.6}$$

L'ampiezza dello step è un punto cruciale per non finire facilmente in overfitting. Per garantire uno step ragionevole si utilizzano tecniche che va-

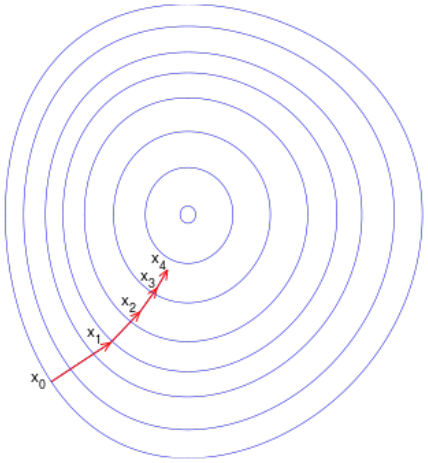

Figura 4.2: Illustrazione del gradiente discendente su una serie di step

lutano il valore di loss in relazione con il passo che il gradiente ha effettuato. Vedremo più avanti una funzione nota utlizzata nel modello costruito.

Backpropagation L'introduzione del *gradient descent* che ripetutamente computa il gradiente e restituisce i parametri aggiornati, introduce il concetto della *backpropagation*. Come espletato dal nome stesso, questo è il processo per cui, una volta ottenuti i nuovi parametri W grazie all'ottimizzazione, "torna indietro" aggiornando i valori stessi per poter iniziare un nuovo ciclo di training (figura 5.3).

Una volta introdotti e spiegati i concetti visti fino ad ora, possiamo entrare nel merito dell'implementazione del modello per la classificazione delle immagini. Parliamo di **Deep Learning** e **reti convoluzionali**.

4.2 Deep Learning e reti convoluzionali

Con il termine *Deep Learning* (DL) si denotano reti "profonde" composte da più livelli organizzati gerarchicamente [35].

Il concetto fondamentale è appunto il ***livello***. Torniamo a vedere la figura 5.3: più livelli sono connessi tra loro attraverso i nodi della rete (chiamati *neuroni*). Una rete, per essere considerata "profonda", ha bisogno almeno di 4 livelli: uno di **input**, uno di **output** e due **nascosti** (*hidden layer*). Quindi, ad esempio, il grafico 5.4 non rappresenta una rete neurale profonda. L'organizzazione gerarchica consente di condividere e le riusare informazioni. Lungo la gerarchia è possibile selezionare *feature* specifiche e scartare dettagli inutili (al fine di massimizzare l'apprendimento).

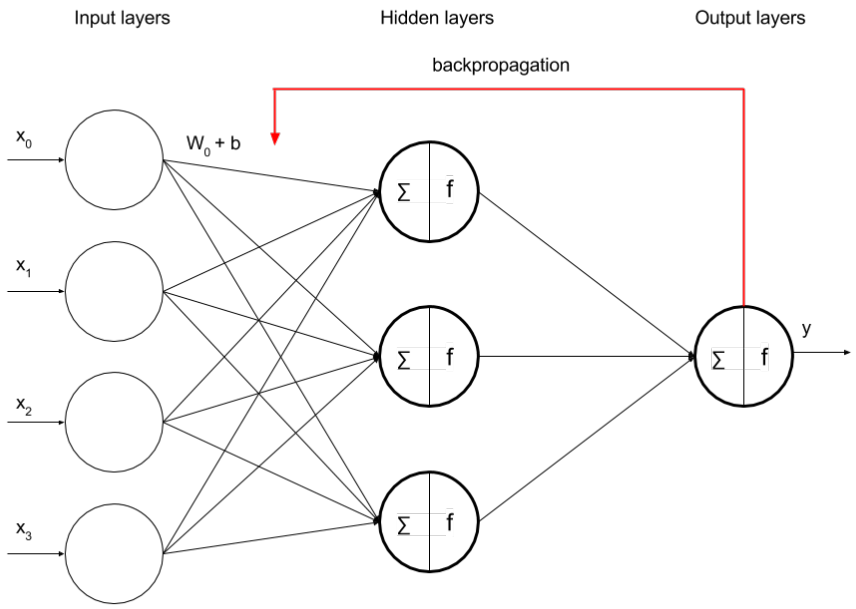

Figura 4.3: Grafico di una rete neurale con relativa backpropagation

Oltre la profondità, uno dei fattori principali da tenere in considerazione è il **numero dei pesi**, ovvero la quantità di parametri che il modello deve apprendere: più pesi comportano più complessità della fase di *training*. Allo stesso tempo anche un numero troppo elevato di neuroni (e connessioni) rende il trasporto delle informaioni molto più costoso.

4.2.1 Convolutional neural network

Le reti neurali convoluzionali fanno la propria comparsa nel 1998, introdotte da Yann LeCun [36], direttore della sezione di ricerca sull' intelligenza artificiale di Facebook. Le reti convoluzionali sono simili ad altri tipi di modelli di deep learning, ma presentano due principali differenze:

- ***processing* locale** i neuroni sono connessi solo localmente ai neuroni del livello precedente. Ogni neurone esegue quindi un'elaborazione locale. Riduzione numero di connessioni.

- **pesi condivisi** i pesi sono condivisi a gruppi. Neuroni diversi dello stesso livello eseguono lo stesso tipo di elaborazione su porzioni diverse dell'input. Riduzione del numero di neuroni.

Come detto sopra, la rete riceve un'immagine in input, e la trasforma attraverso una serie di *hidden layers*. Ogni livello è composto da un serie di neuroni dove ognuno di essi è connesso a tutti quelli del precedente livello, ma nodi dello stesso *layer* non condividono alcun legame tra loro e sono completamente indipendenti.
L'ultimo livello è l'output, e nella classificazione rappresenta la classe predetta corrispondente all'immagine.

Al giorno d'oggi l'uso di reti neurali convoluzionali si è diffuso soprattutto nelle grandi aziende, dove le immagini sono essenziali o contribuiscono in larga parte alla mole di traffico che si sposta. Ad esempio Facebook utilizza le reti convoluzionali per i propri algoritmi di *tagging* delle foto, o per filtrare immagini inappropriate. La classificazione di un'immagine è possibile solo grazie al cosiddetto *image processing*: l'immagine viene suddivisa in array di pixel e gli viene assegnata un'etichetta. Si può riassumere il processo di riconoscimento come segue:

- **Input**, costituito da N immagini rappresentanti il *training set*, ognuna etichettata con una delle K differenti classi.
- **Learning**, processo fondamentale con cui riusciamo a classificare le immagini. Si usa il *training set* per imparare le associazioni tra immagine e classi.
- **Evaluation**, alla fine del processo di *learning*, si deve valutare la qualità del modello in temini di accuratezza e minimizzazione della *loss*. Per fare ciò utilizziamo un altro insieme di M immagini (con $M = 10\%$ di N) chiamato *test set*, composto di dati che il modello istruito non ha mai visto, in modo da poter valutare le previsioni effettuate ed eventualmente operare cambiamenti sui pesi per il *training*.

Quali sono le caratteristiche principali di una rete convoluzionale ?
Esplicitamente progettata per processare immagini, una *convolutional neural network* (CNN) è composta da una gerarchia di livelli, in cui il *layer* di input è direttamente collegato ai *pixel* dell'immagine, mentre gli ultimi livelli sono chiamati *fully-connected* e operano come classificatori; i livelli intermedi utilizzano connessioni locali e pesi condivisi per il processo di convoluzione dell'immagine e apprendimento. I neuroni si organizzano in tre dimensioni, (*width*, *height*, *depth*), evitando che l'immagine debba essere modificata per poter essere acquisita.

Altra peculiarità di queste reti è la tipologia dei livelli:

- INPUT per l'acquisizione dell'immagine RGB (3 canali) o CYMK (4 canali)
- CONVOLUTION layer per l'applicazione dell'operazione di *convoluzione* sull'input (immagine), passando il risultato al layer successivo.

La convoluzione è l'approssimazione della risposta di ogni neurone rispetto a ciò che "vede". La convoluzione opera semplificando l'immagine, riducendo il numero dei parametri da considerare e incrementando la generalizzazione.

- RELU (REctified Linear Units) layer è importante perché applica una semplice funzione di attivazione $f(x) = max(0, x)$ dove x è l'input di un nodo. Questa funzione permette di porre una soglia per cui non possano esserci valori di input negativi. A beneficiarne è soprattutto l'operazione di ricerca del gradiente, oltre ad una velocizzazione dei neuroni che devono effettuare operazioni dispendiose.

- POOLING layer effettua un'operazione di riduzione della dimensione spaziale dell'immagine (altezza, lunghezza), riducendone il volume

- FC (fully connected) layer restituisce la classe predetta, dove la dimensione del risultato sarà sempre [1x1xN], dove N sarà il numero di classi prestabilite.

Una rete convoluzionale trasforma un'immagine livello dopo livello, passando dai valori originali dei pixel al risultato finale della classe associata.

Convoluzione La convoluzione è una delle più importanti operazioni di *image processing* attraverso la quale si applicano filtri digitali. Un filtro digitale (una piccola maschera di pesi chiamata anche *kernel*) è fatta scorrere sulle diverse posizioni di input; per ogni posizione viene generato un valore di output, eseguendo il prodotto scalare tra la maschera e la porzione dell'input coperta (entrambi trattati come vettori).

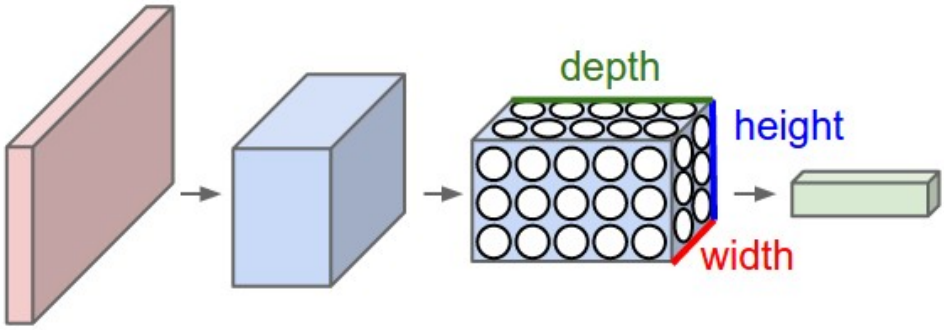

Figura 4.4: Schema di convoluzione

Osserviamo la figura 5.5. Consideriamo i pixel come neuroni e le due immagini di input x e output y come livelli di una rete. Dato un filtro 3x3, se colleghiamo un neurone ai nove neuroni che esso "copre" nel livello

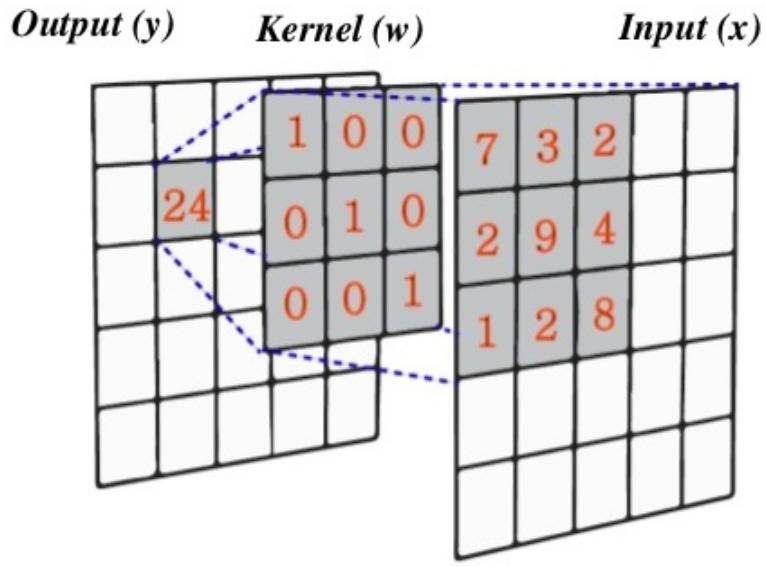

Figura 4.5: Operazione di filtraggio con un kernel 3x3

precedente, e utilizziamo i pesi del filtro w come pesi delle connessioni, notiamo che è ogni neurone stesso ad eseguire di fatto la convoluzione.

Quando un filtro viene fatto scorrere sul volume di input, invece di spostarsi con passi unitari (di 1 neurone) si può utilizzare un passo (*stride*) maggiore. Questa operazione riduce la dimensione delle del volume di output e conseguentemente il numero di connessioni. Di solito sui livelli iniziali si opera uno *stride* di 2 o 4 (a seconda della dimensione dell'immagine) ottenendo un elevato guadagno a discapito di una leggera penalizzazione dell'accuratezza.

ReLU Nelle reti neurali standard, la funzione di attivazione più utilizzata è la **sigmoide**. Nelle reti profonde, l'utilizzo di questa funzione è particolarmente problematico per la *backpropagation*: la derivata della sigmoide è minore di 1, la retropropagazione dei pesi si effettua con derivate a catena, la moltiplicazione di termini minori di 1 allontanerebbe il nostro gradiente dai valori ottimali per cui stiamo facendo il *training*. Si otterrebbe l'effetto opposto di ciò che vogliamo.

Ecco perché si utilizza la funzione di attivazione chiamata ***Relu* (*Rectified Linear*)**. Come vediamo dalla figura 5.6 la derivata vale 0 per valori negativi o nulli. In questo modo è anche possibile scartare (***dropout***) i neuroni che risultano "spenti" (cioè con valore negativo o zero).

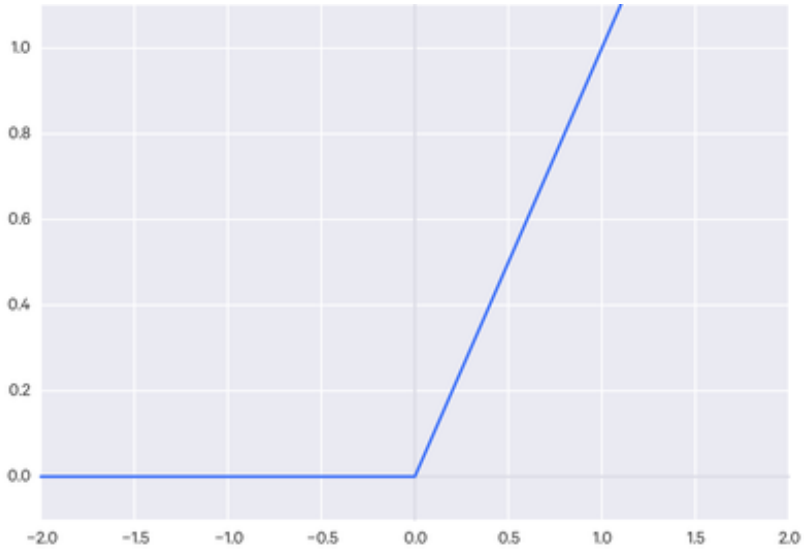

Figura 4.6: ReLU activation

Pooling Un livello di *pooling* esegue un'aggregazione delle informazioni nel volume di input, generando come risultato una mappa dell'immagine di dimensione inferiore. l'obiettivo è mantenere le caratteristiche importanti rispetto a semplici trasformazioni dell'input. Le operazioni di *pooling* possono essere la **media** (*Avg*) o il **Massimo** *Max*.

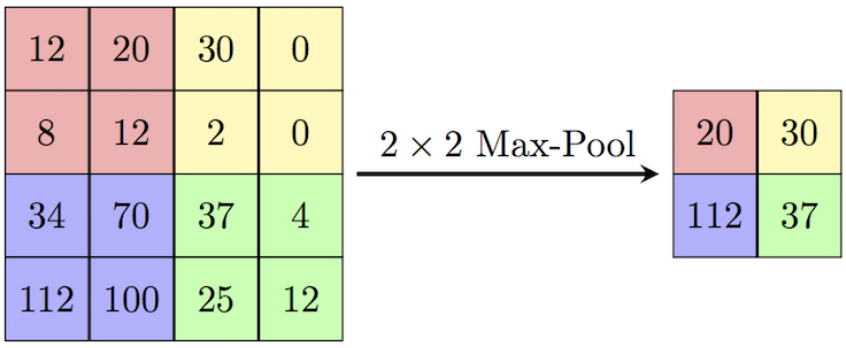

Figura 4.7: Operazione di Max Pooling

SoftMax Nelle CNN si utilizza spesso un livello finale chiamato *Softmax layer*.
È costituito da s neuroni (uno per ogni classe) connesso a tutti i neuroni del livello precedente (*fully-connected*). La differenza con i livelli precedenti è che la funzione di attivazione non è più la $ReLU$ ma ogni k-esimo neurone

utilizza la funzione $Softmax$, dove i valori sono interpretati come probabilità e la loro somma è 1.

Matematicamente la funzione è così descritta:

$$P(y_i|x_iW) = \frac{e^f_{y_i}}{\sum_{n=1}^{s} e^f_{y_i}} \tag{4.7}$$

dove il livello di attivazione del singolo neurone è rapportato alla sommatoria del livello di attivazione di tutti i neuroni finali.

Capitolo 5

Implementazione del modello

5.1 Creazione del modello

Dopo una doverosa premessa teorica, veniamo all'implementazione del classificatore di immagini.
La costruzione del modello si rifà ad un problema già affrontato da altri sviluppatori e studiosi di *Deep Learning*. Il lavoro si basa sui concetti divulgati in una pubblicazione del Dipartimento di Informatica e Ingegneria della *American University of Cairo*, *Applying deep learning to classify pornographic images and videos* [22]. Ricordiamo che il nostro obiettivo è quello di caratterizzare il contenuto del traffico di rete, e questo modello convoluzionale ci servirà per catalogare le immagini in possibili scenari reali.

L'architettura che sta alla base del classificatore si basa su uno dei principali algoritmi applicati per il riconoscimento di immagini in reti convoluzionali: ***AlexNet*** [23].

Algoritmo AlexNet modificato Sviluppato nel 2012 da Alex Krizhevsky dell'Università di Toronto, in pochi anni è divenuto uno dei principali algoritmi per sviluppare modelli di classificazione d'immagine. E' composto da otto livelli: i primi cinque sono convoluzionali, gli ultimi tre sono *fully connected*. L'ultimo livello è quello di output con gli N possibili valori a seconda della classe predetta. L'algoritmo punta a massimizzare la catalogazione delle immagini su più classi (AlexNet è stato sviluppato per predire 1000 classi diverse di immagini). Il primo livello prende in input immagini di dimensione [224x224x3] con 96 *kernel* (filtri) per la convoluzione dell'immagine, trasformandola in una di dimensione [11x11x3], che diventerà l'input del secondo livello. Il processo va avanti fino ad arrivare all'ultimo layer *fully connected*, che ha N uscite per il numero di categorie proposte.

L'implementazione del nostro modello presenta alcune differenze rispetto all'algoritmo originale, in modo da avere due categorie finali possibili. Come si nota dalla figura 6.1, si mantengono 5 livelli di neuroni convoluzionali

seguiti da 3 *fully connected* (l'ultimo è il livello di output). Ogni *layer* convoluzionale filtra l'input grazie a kernel con dimensione e passo prestabiliti in fase di inizializzazione dei parametri. Ognuno dei neuroni convoluzionali presenta una funzione di attivazione (ReLU) per velocizzare il processo di *learning*.
L'output dell'ultimo livello è "riempito" con una funzione *softmax* che produce una distribuzione su due possibili classi: *Illecito* o *Lecito*.

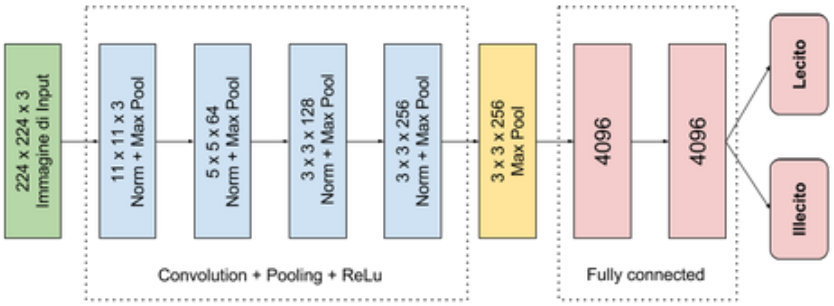

Figura 5.1: Grafico della rete implementata seguendo l'algoritmo AlexNet

Tensorflow Per la rete, ci siamo serviti di una libreria *open source* di *machine learning* sviluppata da Google: ***TensorFlow*** [38]. Tensorflow è una libreria per implementare ed eseguire algoritmi di ML. Un modello epresso usando TensorFlow può essere eseguito con piccole modifiche (o nessuna) su una vasta varietà di sistemi eterogenei, dai dispositivi mobili a sistemi distribuiti con centinaia di macchine GPU collegate tra loro. Questa elasticità permette appunto di poter affrontare molte problematiche, compresa l'implementazione di algoritmi di *deep learning* costruendo reti neurali profonde utilizzando le API fornite [39].

- Una computazione con TensorFlow è descritta da un **grafo**, che è composto da una serie di **nodi**. I nodi del grafo sono implementati per mantenere aggiornate e persistenti le informazioni acquisite durante la fase di *training*. Ogni nodo rappresenta l'istanza di una operazione matematica. I valori che fluiscono nel grafo lungo i collegamenti sono chiamati **tensori**, vettori multimediali complessi, contenenti le informazioni che si passano i nodi dei vari livelli. Possiamo dire che i tensori sono la rappresentazione dei dati memorizzati dai nodi durante la computazione del grafo.
- Il nucleo della libreria è scritto in `C++`, ma la libreria presenta API scritte anche in *Python*. Dalle ultime versioni è stato esteso il supporto anche per *Java* e *Go*.

- L'interazione con il sistema TensorFlow avviene attraverso una **sessione**. Ogni computazione che vogliamo eseguire nel grafo deve essere "lanciata" all'interno di una sessione. Possono esserci più sessioni legate a più operazioni che si devono svolgere all'interno del grafo: per ogni computazione deve essere specificata una sessione diversa.

5.2 Costruzione della rete ConvNet

Il progetto è così suddiviso:

- `my_alexnet_cnn.py` rappresenta il nucleo del programma, in cui vi è l'implementazione dell'algoritmo AlexNet e le due funzioni di *training* e *testing*;
- `Dataset.py` implementa la parte relativa al *dataset*, in particolare per quanto riguarda il *preprocessing* delle immagini;
- `train_dataset` la cartella contenente le sottocartelle con le immagini da sottoporre alla fase di *training*;
- `test_dataset` la cartella contenente le sottocartelle con le immagini da sottoporre alla successiva fase di *testing*;

5.2.1 Preprocessing delle immagini

Prima di analizzare in dettaglio l'implementazione della rete e descriverne il funzionamento, parliamo di un'operazione preliminare necessaria. Il ***preprocessing*** delle immagini è invocato nel `main()` e suddiviso tra immagini d *training* e di *testing*.

```
# PREPROCESSING TRAINING
 if args.which == 'preprocessing_training':
        if args.shuffle:
            l = [i for i in
                Dataset.loadDataset('images_dataset.pkl')]
            np.random.shuffle(l)
            Dataset.saveShuffle(l)
        else:
            Dataset.saveDataset(TRAIN_IMAGE_DIR, args.file)

 # PREPROCESSING TEST
 elif args.which == 'preprocessing_test':
        Dataset.saveDataset(TEST_IMAGE_DIR, args.test)
```

L'operazione per il *training* è invocata nel modo seguente: `python my_alenxet_cnn.py preprocessing_training -s` . La prima invocazione è sulla funzione `saveDataset` che si trova nel file `Dataset.py`:

```
def saveDataset(image_dir, file_path):
    with gzip.open(file_path, 'wb') as file:
        for img, label in convertDataset(image_dir):
            pickle.dump((img, label), file)
```

La funzione prende il path di ogni cartella e converte ogni immagine di formato `.jpg` o `.jpeg` in un flusso di *byte* associando ogni immagine alla sua etichetta, utilizzando la libreria `pickle` [40]. Questo modulo di Python prende un oggetto e lo converte appunto in un *byte stream* (`pickle.dump`); effettua l'operazione opposta se invocata la funzione (`pickle.load`), come si vede nel seguente metodo `loadDataset`:

```
def loadDataset(file_path):
   with gzip.open(file_path) as file:
      while True:
         try:
               yield pickle.load(file)
           except EOFError:
               break
```

Fondamentale è capire come le immagini vengono associate ad ogni etichetta. La funzione `saveDataset`, per ogni cartella, invoca `convertDataset`. Dopo aver aperto una sessione *TensorFlow*, si controlla che l'estenzione sia `.jpg` o `.jpeg`, si decodifica l'immagine con la funzione della libreria `tf.image.decode_jpeg(img_bytes, channels=3)` e si "standardizza" la dimensione dandole un formato univoco, in modo che l'immagine di ingresso sia conforme con l'algoritmo AlexNet, che richiede immagini di dimensione [224x224x3]. A questo punto si effettua lo *shape* sull'immagine, ovvero si "appiattisce" la foto per poterla convertire con `pickle`, associandole l'etichetta della propria classe. Vediamo il pezzo di codice della funzione `convertDataset` che esprime le operazioni sopra descritte:

```
   [...]
   img_bytes = tf.read_file(img_path)
   img_u8 = tf.image.decode_jpeg(img_bytes, channels=3)
   img_u8_eval = session.run(img_u8)
   image = tf.image.convert_image_dtype(img_u8_eval, tf.float32)
   img_padded_or_cropped =
       tf.image.resize_image_with_crop_or_pad(image, IMG_SIZE,
       IMG_SIZE)
   img_padded_or_cropped = tf.reshape(img_padded_or_cropped,
       shape=[IMG_SIZE * IMG_SIZE, 3])
   yield img_padded_or_cropped.eval(session=session), label_i
```

Prima di passare all'implementazione della rete, è importante ricordare che le immagini non vengono caricate sempre nel solito ordine. Dopo la

conversione in `pickle` grazie all'opzione `-s`, è possibile effettuare la ridistribuzione casuale (*shuffle*) di tali immagini in modo da migliorare ogni volta la fase di apprendimento che, altrimenti, potrebbe essere meno veritiero se le immagini fossero acquisite ogni volta nel solito ordine. Lo *shuffle* viene applicato al file `pickle` già convertito per rendere il processo più veloce.

5.2.2 Implementazione dell'algoritmo AlexNet

Gli esperimenti effettuati sulla rete dopo ci hanno portato a discostarci leggermente dall'algoritmo originale di AlexNet, e i vari test ci hanno permesso di trovare la migliore configurazione possibile dei parametri per ottenere la più alta accuratezza possibile.
Il modello è creato come istanza di un oggetto Python, dove il metodo "costruttore" `__init__` carica il *dataset*, inizializza le variabili per i pesi e gli errori (*weights* e *biases*), dichiara i cosiddetti *placeholder*, strutture che vengono "riempite" con i dati che si muovono nel grafo, e crea un punto di *checkpoint* per la sessione di *training*.
Il seguente codice mostra le sezioni spiegate:

- **caricamento dataset**

```
class ConvNet(object):
    ## Constructor to build the model for the training ##
    def __init__(self, **kwargs):
    ...
    if(self.dataset_training != False):
       self.train_imgs_lab =
           Dataset.loadDataset(self.dataset_training)
    else:
        self.test_imgs_lab = Dataset.loadDataset(self.dataset_test)
```

- **inizializzazione variabili *weights* e *biases***

```
  self.weights = {
     'wc1': tf.Variable(tf.random_normal([11, 11, n_channels,
         BATCH_SIZE], stddev=std_dev)),
       'wc2': tf.Variable(tf.random_normal([5, 5, BATCH_SIZE,
           BATCH_SIZE*2], stddev=std_dev)),
       'wc3': tf.Variable(tf.random_normal([3, 3, BATCH_SIZE*2,
           BATCH_SIZE*4], stddev=std_dev)),
       'wc4': tf.Variable(tf.random_normal([3, 3, BATCH_SIZE*4,
           BATCH_SIZE*4], stddev=std_dev)),
       'wc5': tf.Variable(tf.random_normal([3, 3, BATCH_SIZE*4,
           256], stddev=std_dev)),
```

```
        'wd': tf.Variable(tf.random_normal([2*2*256, 4096])),
        'wfc': tf.Variable(tf.random_normal([4096, 2*2*256],
            stddev=std_dev)),

        'out': tf.Variable(tf.random_normal([2*2*256, n_classes],
            stddev=std_dev))
    }

self.biases = {
    'bc1': tf.Variable(tf.random_normal([BATCH_SIZE])),
    'bc2': tf.Variable(tf.random_normal([BATCH_SIZE*2])),
    'bc3': tf.Variable(tf.random_normal([BATCH_SIZE*4])),
    'bc4': tf.Variable(tf.random_normal([BATCH_SIZE*4])),
    'bc5': tf.Variable(tf.random_normal([256])),

    'bd': tf.Variable(tf.random_normal([4096])),
    'bfc': tf.Variable(tf.random_normal([2*2*256])),

    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

Questo pezzo di codice dà molte informazioni discusse anche nella parte teorica introduttiva.
Il numero delle variabili utilizzate corrisponde al numero dei livelli utilizzati nella rete (8), illustrando quanti siano quelli convoluzionali (i primi 5) e quali i *fully-connected* (2 più l'ultimo `out` che dà il risultato della rete).

Una *Variable* è uno speciale operatore che ritorna il controllo ad un tensore mutabile che rimane in vita durante l'esecuzione del grafo. In TensorFlow tipicamente i parametri del modello sono salvati in un tensore contenuto nella variabile, in modo che siano aggiornati durante l'esecuzione del grafo.

Guardiamo la prima variabile: `tf.Variable(tf.random_normal([11, 11, n_channels, BATCH_SIZE], stddev=std_dev))`.
Il codice ci dice che la variabile dei pesi del primo livello convoluzionale prende in ingresso un'immagine [224 x 224 x 3] (di default) e vi applica un filtro di dimensioni [11 x 11 x 3 x BATCH_SIZE]. Il valore di `BATCH_SIZE` è stato impostato a 64, e rappresenta il numero di immagini processate ad ogni ciclo di training. Questa procedura è molto utilizzata nelle reti convoluzionali, in quando analizzare una singola immagine alla volta allungherebbe a dismisura i tempi di *training*, specie per *dataset* molto grandi.

```
self.img_pl = tf.placeholder(tf.float32, [None, n_input,
    n_channels])
self.label_pl = tf.placeholder(tf.float32, [None, n_classes])
```

Un *placeholder* è una semplice variabile che verrà assegnata ad un dato. Questo permette a TensorFlow di creare il proprio grafo e le proprie opera-

zioni senza bisogno dei dati. In seguito, quando la computazione è lanciata, i *placeholder* verranno riempiti con i dati di input che si muovono nel grafo. In questo caso ha senso definire le immagini e le etichette come *placeholder*, poiché esse variano ad ogni ciclo del grafo.

- **implementazione dei livelli**

I livelli della rete sono implementati all'interno del metodo `alex_net_model`, che viene chiamato sia nella fase di *training* che di *testing*. I parametri di questo metodo sono l'immagine, le variabili `weights` e `biases` definite nel costruttore, e i valori di **dropout** per determinare la soglia per cui un neurone debba essere considerato spento e quindi cancellato.

I primi 5 layer sono convoluzionali

```
# Convolutional Layer 1
 conv1 = self.conv2d('conv1', _X, _weights['wc1'],
     _biases['bc1'], s=4)
 print "conv1.shape: ", conv1.get_shape()
 # Max Pooling
 pool1 = self.max_pool('pool1', conv1, k=3, s=2)
 print "pool1.shape:", pool1.get_shape()
 # Apply Normalization
 norm1 = self.norm('norm1', pool1, lsize=4)
 print "norm1.shape:", norm1.get_shape()
 # Apply Dropout
 dropout1 = tf.nn.dropout(norm1, input_dropout)
```

La prima è un'invocazione del metodo `conv2d` così definito

```
def conv2d(self, name, l_input, w, b, s):
   return tf.nn.relu(tf.nn.bias_add(tf.nn.conv2d(l_input, w,
       strides=[1, s, s, 1], padding='SAME'), b), name=name)
```

Si applica la convoluzione sull'immagine di input `l_input`, passando ai neuroni la variabile dei pesi per il primo livello di convoluzione `wc1`, filtrando la stessa immagine con un passo di dimensione [1,4,4,1] (cioè la finestra si muove di 4 pixel in 4 pixel). Ogni covoluzione è attivata con la funzione `tf.nn.relu` (ReLU *activation*).

Finita la convoluzione si applica la funzione di *pooling*. In questo caso `max_pool` ci dice che è un'operazione di massimo applicata ad un filtro 3x3 (`k=3`) con passo di 2 pixel (`s=2`).

Questo produce un output dell'immagine ridotta, che verrà normalizzata grazie alla funzione `norm`, definita come

```
def norm(self, name, l_input, lsize):
   return tf.nn.lrn(l_input, lsize, bias=1.0, alpha=2e-05,
       beta=0.75, name=name)
```

`lrn` sta per *local response normalization*, e si utilizza per prevenire la saturazione dei neuroni quando gli input hanno diversa dimensione. Si cerca di dare una generalizzazione ai nodi del grafo.

I primi 4 livelli di convoluzione adottano l'operazione di *dropout* per eliminare eventuali neuroni disattivati se il risultato della funzione non supera una certa soglia. Per il livello di input il valore è posto a 0.8, mentre per i livelli intermeti a 0.5.
Rispetto all'algoritmo orginale, la nostra implementazione prevede le operazioni di *pooling* e normalizzazione anche per il terzo e il quarto livello di convoluzione, cosa non prevista per l'originale algoritmo AlexNet. Si sono fatti diversi test a riguardo e l'accuratezza nella fase di *testing* diminuiva molto in assenza di questa modifica. Cioò può essere causato dalla qualità delle immagini e dalla loro grandezza iniziale. Questa modifica ci ha permesso invece di ottenere i migliori risultati.

Gli ultimi tre livelli sono di tipo *fully-connected*. La presenza di tale livello è necessaria poiché è un modo veloce per imparare combinazioni non lineari che possono essere non riconosciute durante la fase di convoluzione, la quale, modificando l'immagine nello spazio, cerca *feature* di alto livello. Possiamo dire che i *fully-connected layers* osservano le caratteristiche che possono venire fuori dall'operazione di convoluzione dell'immagine.
Ogni livello di questo genere, in sostanza, presenta una funzione di attivazione ReLU

```
# Relu activation
fc1 = tf.nn.relu(tf.matmul(dense, _weights['wd']) + _biases['bd'],
    name='fc1')
dropout7 = tf.nn.dropout(fc2, hidden_dropout)
```

L'ultimo livello, quello di output, genera una probabilità delle classi predette. In sostanza applica la formula $Y = Wx + b$ vista nel capitolo precedente quando si parlava di classificazione.

```
# Output, class prediction LOGITS
  out = tf.matmul(dropout7, _weights['out']) + _biases['out']

# returns the Logits to be passed the Softmax for the PREDICTION
  return out
```

- **Training**

Una volta completato il *preprocessing* delle immagini, si passa alla fase più lunga e importante, il *training*. Da riga di comando si invoca il seguente comando: `python my_alexnet_cnn.py train -lr [...] -e [...] -s [...]` in cui i parametri da riga di comando sono rispettivamente il valore ***learning_rate*** (valore da cui l'ottimizzatore parte per minimizzare la loss e

trovare la massima accuratezza possibile), il numero di **epoche** (i cicli che il metodo deve effettuare), e il *display step* (le epoche da visualizzare e salvare in un apposito file di `log`.

```
# Method for training the model and testing its accuracy
def training(self):
   # Launch the graph
    with tf.Session() as sess:

      ## Construct model: prepare logits, loss and optimizer ##

      # logits: unnormalized log probabilities
      logits = self.alex_net_model(self.img_pl, self.weights,
          self.biases, self.keep_prob_in, self.keep_prob_hid)

      # loss: cross-entropy between the target and the softmax
          activation function
      loss =
          tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,
          labels=self.label_pl))

      tf.summary.scalar("cross-entropy_for_loss", loss)

      # optimizer: find the best gradients of the loss with respect
          to each of the variables
      train_step =
          tf.train.AdamOptimizer(learning_rate=self.learning_rate,
          epsilon=0.1).minimize(loss)
```

In questo primo pezzo di codice, dopo l'inizializzazione della sessione TensorFlow, si chiama il metodo `alex_net_model` illustrato nella sezione precedente, dopodiché abbiamo una chiamata di funzione complessa:

`loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=self.label_pl))`

Ricordandoci lo schema di classificazione spiegato nel capitolo 5, una volta ottenute le probabilità non-normalizzate (*logits*) applichiamo la funzione di *softmax* a tali risultati, in modo da ottenere valori probabilistici tra 0 e 1. Tensorflow mette a disposizione una funzione che applica non solo la softmax, ma calcoli direttamente anche il valore dell'errore per tale funzione (*cross-entropy loss*), prendendone la media con `tf.reduce_mean`.
Essendo nella fase di *training* dobbiamo minimizzare al massimo il valore della *loss*, ed è per questo che ci interessa tale variabile.

La *loss* si minimizza grazie ad un **ottimizzatore**, che cerca il miglior gradiente partendo dal valore del *learning_rate* passato come argomento. L'ottimizzatore usato è l'*AdamOptimizer* [41], uno dei più utilizzati per le reti convoluzionali, in quanto, a differenza di altri, l'algoritmo utilizzato converge più velocemente grazie ad uno *step* maggiore, in modo da velocizzare

il raggiungimento del migliore valore di accuratezza. Di contro richiede più risorse computazionali ad ogni passo di apprendimento.

Prima di invocare il ciclo di *training*, salviamo la lista dei valori corretti per la classe di appartenenza di ogni immagine, e creiamo il tensore per il calcolo dell'accuratezza

```
# list of booleans
  correct_pred = tf.equal(tf.argmax(logits,1),
      tf.argmax(self.label_pl, 1))
# [True, False, True, True] -> [1,0,1,1] -> 0.75
  accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

A questo punto inizializziamo tutte le variabili dichiarate fino ad ora (`init = tf.global_variables_initializer()`, definiamo un sottogruppo di immagini per la validazione dell'accuratezza (`validation_imgs_batch`, e la sessione per il *training* può essere lanciata.

```
# Run for epoch
  for epoch in range(self.max_epochs):
   log.info("Epoch %s" % epoch)
   self.train_imgs_lab = Dataset.loadDataset(self.dataset_training)
       # necessary for the yeld

   # Loop over all batches
   for step, elems in
       enumerate(self.BatchIteratorTraining(BATCH_SIZE)):
      print "step = %d" % step
      # from iterator return batch lists
      batch_imgs_train, batch_labels_train = elems
      _, train_acc, train_loss, summary_op = sess.run([train_step,
          accuracy, loss, merged_summary_op],
          feed_dict={self.img_pl: batch_imgs_train, self.label_pl:
          batch_labels_train, self.keep_prob_in: 1.0,
          self.keep_prob_hid: 1.0})

      summary_writer.add_summary(summary_op, epoch * step + i)

      if step % self.display_step == 0:
         log.info("Training Accuracy = " +
             "{:.5f}".format(train_acc))
         log.info("Training Loss = " + "{:.6f}".format(train_loss))

   print "Optimization Finished!"
```

Il modello impara iterando `max_epochs` volte (opzione `-e` da riga di comando) e, ad ogni ciclo, estrae un gruppo di coppie (immagine, *label*) di dimensione (`BATCH_SIZE`), suddividendo poi la parte dati dell'immagine con la sua etichetta della classe. Queste due liste create, `batch_imgs_train` e

`batch_labels_train` sono passate al metodo `sess.run`, e per ogni immagine è calcolata l'accuratezza e la loss (`train_acc` e `train_loss`).
Alla fine del processo la sessione del modello viene salvata in una cartella, in modo tale da poter ricaricare il modello nella fase di *testing* senza bisogno di effettuare di nuovo questa procedura.

- Prediction

La fase di *testing* è implementata nel metodo chiamato `prediction`.
La funzione chiama il metodo `alex_net_model` per costruire l'oggetto modello (che chiameremo `pred`) e, una volta caricata la sessione con tutti i parametri del modello, analizza le immagini salvate nella cartella *test_dataset* ed effettua la predizione.

```
batch_imgs_test, batch_labels_test = elems

y_pred = sess.run(y_p, feed_dict={self.img_pl: batch_imgs_test,
    self.keep_prob_in: 1.0, self.keep_prob_hid: 1.0})
list_pred_total.extend(y_pred)

y_true = np.argmax(batch_labels_test,1)
list_true_total.extend(y_true)
```

Come si vede le etichette delle immagini di test non sono passate nella fase di `run`, ma verranno salvate in una lista `list_true_total` che sarà confrontata con `list_pred_total` per calcolarne l'accuratezza.

Per poter completare il lavoro efficientemente, la rete è stata fatta girare su una macchina GPU `Tesla K-80` concessa dalla società `Nvidia`, che ci ha gentilmente offerto tale supporto tecnico su formale richiesta.

Capitolo 6

Validazione del lavoro

In questo capitolo valideremo il lavoro svolto, parlando sia dei risultati del dissector creato su nDPI, sia di quelli ottenuti testando le immagini nel modello di rete neurale creato. Verrà presentato il prototipo di un framework in cui si utilizza la libreria nDPI e la rete convoluzionale creata, allo scopo di illustrare un possibile caso d'uso.

6.0.1 Dati in ingresso per catalogare QUIC su nDPI

Per poter validare il *dissector* implementato per il protocollo QUIC, si sono utilizzate varie sessioni di traffico *live*. Gli output per testare la classificazione dei protocolli di nDPI sono i risultati della demo creata appositamente per testare la libreria: `ndpiReader`.

Il primo test è stato effettuato su una sessione di traffico YouTube, di durata 124.5 secondi, contenente 30246 pacchetti per un totale di 31459234 byte (poco più di 30 Mbyte). Le figure 7.1 e 7.2 evidenziano come il maggior numero di flussi Youtube siano rappresentati da sessioni QUIC. Per esempio, due righe di *output* prese dalla figura 7.2 ci mostrano questo riscontro:

```
[proto: 188.124/QUIC.YouTube][8350 pkts/903380 bytes <-> 20930
    pkts/29153360 bytes][Host: r2--sn-bvvbax-4jve.googlevideo.com]

[proto: 188.124/QUIC.YouTube][74 pkts/11892 bytes <-> 130
    pkts/160630 bytes][Host: www.youtube.com]
```

In figura 7.1 possiamo vedere anche le statistiche utilizzate da nDPI per il dissector e la classificazione dei pacchetti: La memoria occupata da nDPI per questa sessione è stata di 111.87 KByte, di cui 1.96 KByte per ogni flusso. Il *throughput* di nDPI (cioè il tasso di pacchetti consegnati con successo tra sorgente e destinazione) è stato di 6.82 Gb/sec.

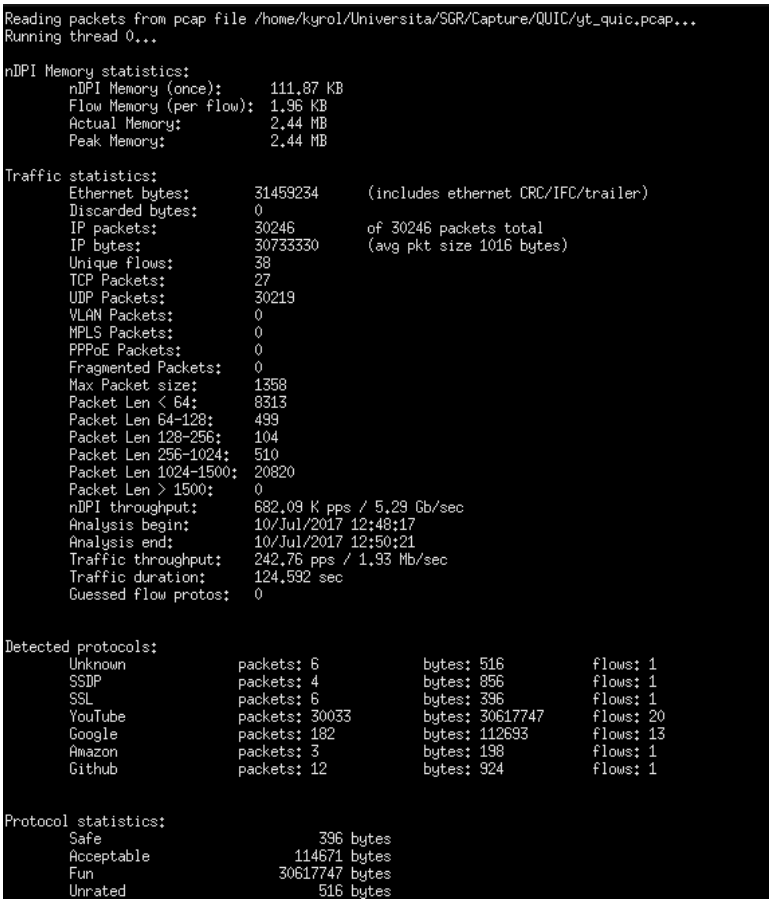

Figura 6.1: Classificazione della sessione di traffico YouTube

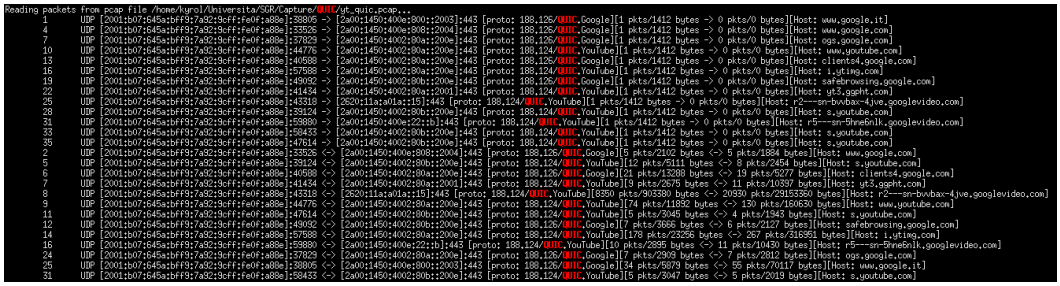

Figura 6.2: Risultati in dettaglio per la classificazione del traffico YouTube

Un secondo test significativo è stato fatto sul traffico `Facebook`, di durata 189.5 secondi, contenente 7991 pacchetti per un totale di 10284008 byte (poco più di 10 Mbyte).

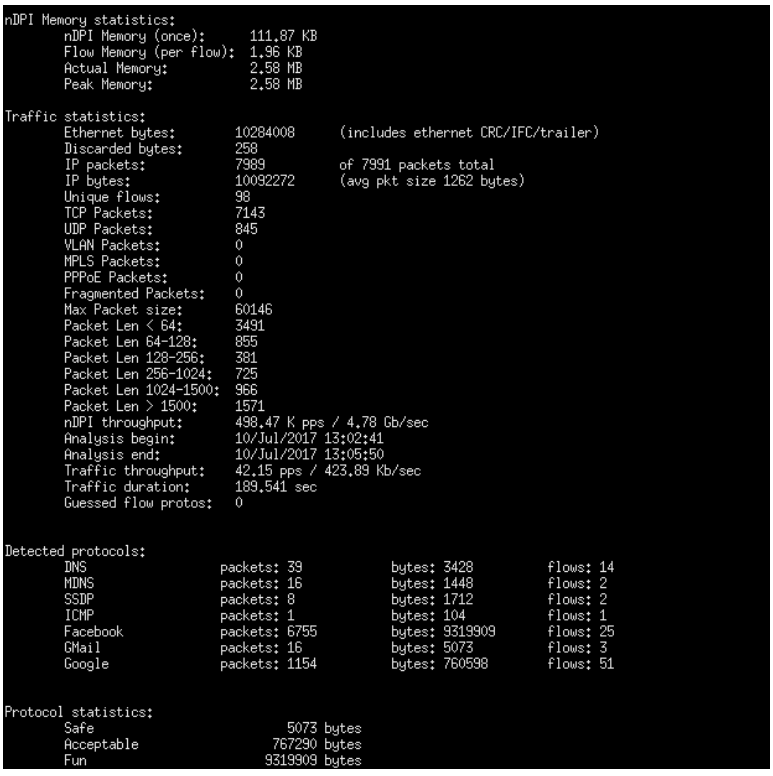

Figura 6.3: Classificazione della sessione di traffico Facebook

Da questi risultati possiamo notare come sia differente l'uso di QUIC in queste due applicazioni. Mentre su `YouTube` il traffico multimediale è prevalentemente trasportato da QUIC (quindi su UDP), per la sessione `Facebook` i flussi per classificare l'applicazione sono estratti da traffico SSL/TLS, mentre QUIC è utilizzato da `Facebook` per contattare i server di `Google`. Cu-

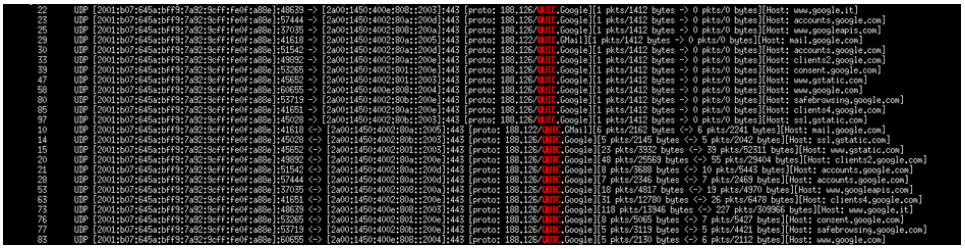

Figura 6.4: Risultati in dettaglio per la classificazione del traffico Facebook

rioso è il fatto che la maggior parte del traffico viaggia su SSL (9319909 bytes contro i 760598 di QUIC), ma il numero di flussi aperti è maggiore per il traffico `Google` rispetto a quello `Facebook`. Questo perché le sessioni QUIC aperte con i server `Google` hanno una durata minore, e per una sessione `Facebook` vengono contattati molti più server Google rispetto ad una sessione `Youtube`.

A causa del massiccio utilizzo di SSL rispetto a `Youtube`, la sessione di `Facebook` diminuisce anche il throughput di nDPI, che è 4.80 Gb/sec.

6.0.2 Dataset per training e testing della rete ConvNet

Per la costruzione del modello della rete neurale è stato utilizzato l' NDPI dataset, ottenuto su richiesta all' NPDI group, Federal University of Minas Gerais (UFMG), Brazil [37]. Il dataset è suddiviso in tre categorie, *Non Porn Difficult* contenente immagini con molta presenza di pelle ma senza nudità, *Porn* contenente immagini di nudo esplicito, e *Non Porn Easy*, con immagini random facilmente classificabili come lecite.

Le immagini del dataset sono state suddivise in tre parti: ***Training set***, ***Validation set*** e ***Test set***.

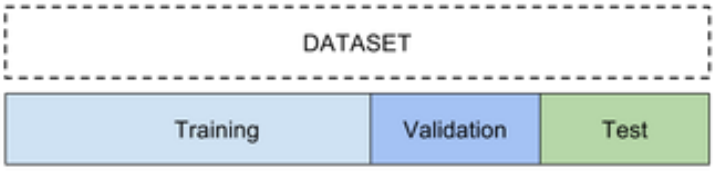

Figura 6.5: Suddivisione del dataset in training, validation e test set

Il *Training set* presenta in totale 12865 immagini, di cui 768 lasciate per il *Validation set*, mentre il *Test set* è composto da 1370 immagini. Il

Validation set è necessario per la fase di *training*, in quanto regolarizza i risultati di accuratezza calcolati sul *Training set*.

Le immagini del dataset, come detto, subiscono una fase di *preprocessing* in cui sono trasformate in byte e compresse in un file *pickle* (*image_dataset.pkl*). La fase di training opera sul file mischiato dopo aver fatto lo *shuffle* del file *pickle* creato. Il nuovo file si chiamerà *images_shuffled.pkl*.

Più grande è il dataset delle immagini, maggiore sarà il file *pickle* creato. In questo caso, con circa 12 K immagini di dimensione variabile, il file *image_dataset.pkl* ha una dimensione di 1.8 GByte.

Un altro problema di spazio è causato dalla cartella in cui è salvato il modello dopo la fase di training. `Tensorflow` salva ogni evento e variabile durante la fase di training del modello, e comporta la creazione di un file di dimensione consistente (ad esempio `events.out.tfevents.1498.R730-gpu`) che occupa 5.7 GByte. Viene salvato anche il file di *checkpoint* del modello, che però ha dimensioni limitate (circa 40 Mbyte).

6.0.3 Qualità dei risultati ottenuti per la rete

I due grafici di figura 6.6 e 6.7 riguardano l'*accuracy* e la *loss*. Si vede come il primo converge abbastanza in fretta verso il miglior risultato possibile (grazie alla scelta dell'ottimizzatore *Adam* durante la fase di training). Allo stesso modo il valore della *loss* diminuisce in modo inversamente proporzionale e coerente con l'aumentare dell'accuratezza. I due grafici sono stati visualizzati con *TensorBoard* [42], una *suite* di TensorFlow per la visualizzazione delle metriche di visualizzazione del grafo.

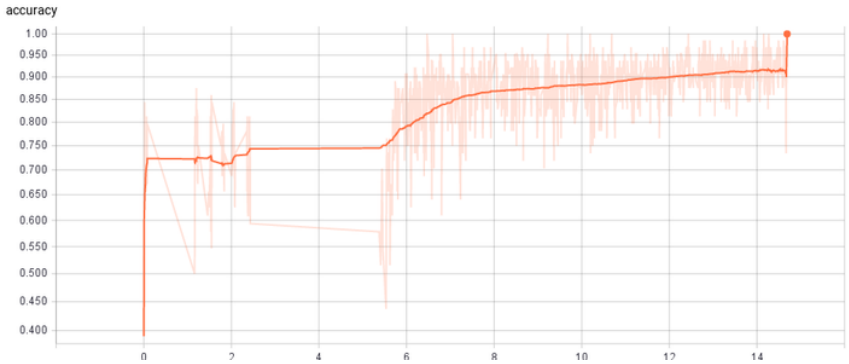

Figura 6.6: *Accuracy improvement* per la fase di training

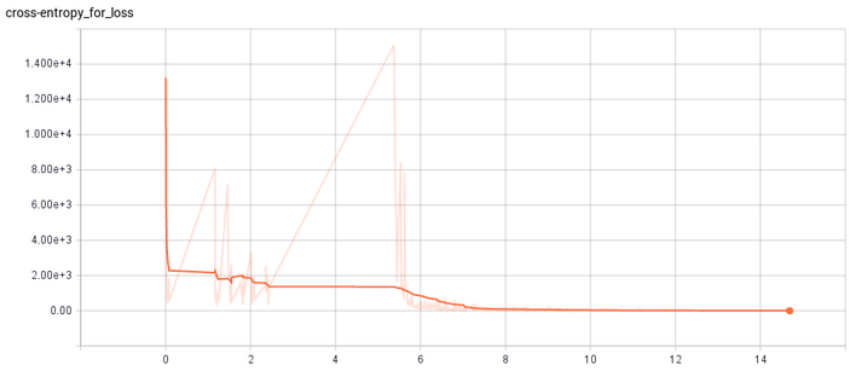

Figura 6.7: *Loss decreasing* per la fase di training

Per il calcolo dell'accuratezza del modello, abbiamo valutato i risultati di tre misurazioni (*precision*, *recall*, *f1-score*) su tre differenti set di parametri (spiegate nel capitolo 4).

Nelle seguenti due tabelle possiamo visualizzare i risultati ottenuti per il *training set* e *testing set*, basati su differenti valori di *learning_rate* e numero di epoche.

Training Set results with epochs = 90			
Learning rate and max epochs	Precision	Recall	F1-Score
l.r. = 0.01	0.23	0.48	0.31
l.r. = 0.001	0.93	0.93	0.93
l.r. = 0.0001	0.96	0.96	0.96

Test Set results with epochs = 90			
Learning rate and max epochs	Precision	Recall	F1-Score
l.r. = 0.01	0.25	0.50	0.33
l.r. = 0.001	0.83	0.83	0.81
l.r. = 0.0001	**0.84**	**0.84**	**0.84**

Prendendo in considerazione il miglior risultato ottenuto, vediamo in dettaglio i risultati per il *Test set*.

Test Set learning rate = 0.0001 max epochs = 90				
	Precision	Recall	F1-Score	Support
Ilecito	0.82	0.86	0.84	685
Lecito	0.86	0.81	0.83	685
avg / total	0.84	0.84	0.84	1370

Confrontando i risultati ottenuti dalla nostra rete possiamo che siamo in linea con quelli dello stato dell'arte.

6.0.4 Prototipo di framework con nDPI e ConvNet

La figura 6.8 illustra il prototipo di un framework che permetta l'estensione di nDPI con il modello di Deep Learning creato. Vediamo l'evoluzione per il caso della caratterizzazione di traffico HTTP.

Una volta che il traffico arriva a nDPI, viene invocato il dissector per HTTP (`http.c`) per il riconoscimento del protocollo. Al suo interno, il dissector effettua il *parsing* del payload, verificando, tra i vari campi, quello che ci indica la presenza o meno di un file ultimediale: `Content-Type`.

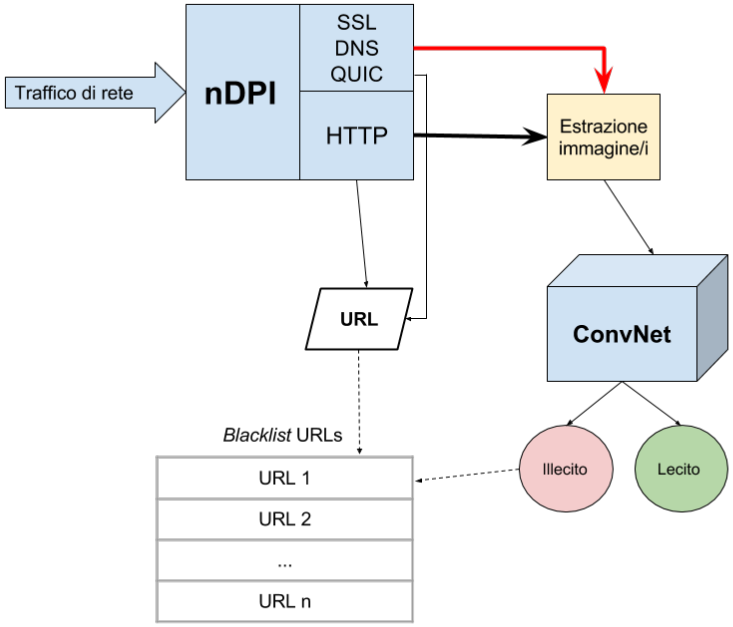

Figura 6.8: Scenario del framework per l'utilizzo di nDPI e ConvNet

```
if(packet->line[packet->parsed_lines].len > 13
   && memcmp(packet->line[packet->parsed_lines].ptr,
       "Content-type:", 13) == 0) { ... }
```

Se questo campo non è presente, il pacchetto è classificato semplicemente come HTTP e poi scartato.
Se invece è presente, si guarda se è della forma `Content-Type: image/jpeg` (il nostor modello processa solo immagini *jpg o jpeg*. Data la presenza dell'immagine, il campo `Content-length` ci indica la dimensione (in byte) del dato, che provvediamo a salvare sul disco (ringrazio Daniele De Sensi per questo pezzo di codice dal progetto *Peafowl* [43]).

```
while(img_file_n < packet->http_url_name.len){
   sprintf(img_name, "jpeg_dump/%s:%"PRIu16".jpeg", src);
   }
```

L'immagine viene salvata in una cartella con il nome dell'*url* estratto dal dissector di HTTP (`packet->http_url_name`). Questo ci servirà per creare la corrispondenza tra immagine e *url*.

A questo punto il dato estratto può essere passato al modello **ConvNet** creato, il quale di default, in ingresso, ridimensionerà l'immagine per

conformarla alle dimensioni scelte, [224x224x3]. La rete prende in ingresso la cartella contenente tutte le sotto-cartelle dei vari `url`, ognuna avente immagini da classificare.

Il modello dovrà essere già stato caricato precedentemente.

```
for dirName in os.listdir(nDPI_IMAGE_DIR):
   path = os.path.join(nDPI_IMAGE_DIR, dirName) # dirname is the URL
      for img in os.listdir(path):
           print "reading image ... "
           ...
           ...
           ...
           # Check if image is a correct JPG file
            if(os.path.isfile(img_path) and
                (img_path.endswith('jpeg') or
                (img_path.endswith('jpg')))):
               # Read image and convert it
               img_bytes = tf.read_file(img_path)
               img_u8 = tf.image.decode_jpeg(img_bytes, channels=3)
                ...
                img_padded_or_cropped =
                    tf.image.resize_image_with_crop_or_pad(image,
                    IMG_SIZE, IMG_SIZE)
            # Run the model to get predictions
               predict = sess.run(prediction, feed_dict={self.img_pl:
                   [img_eval], self.keep_prob: 1.})
               pred_string =
                   (LABELS_DICT.keys()[LABELS_DICT.values().index(predict)])
               print "ConvNet prediction = %s" % pred_string # Print
                   the name of class predicted

            return pred_string, dirname
            else:
               print "ERROR IMAGE"
```

La classificazione è pressoche immediata: se l'immagine sarà etichettata come **illecita**, la rete ritornerà alla libreria nDPI la coppia (*immagine*, *url*). nDPI inserirà l'indirizzo in un'apposita *blacklist* che verrà creata.

La *blacklist* mantenuta sarà utilizzata da applicazioni che usano nDPI per la classificazione, le quali operano nel filtraggio e nella monitorazione del traffico.

Capitolo 7

Conclusioni e sviluppi futuri

Il repentino sviluppo di servizi e applicazioni nelle reti moderne, ha portato la necessità di dare una classificazione qualitativa del traffico. Se prima bastava individuare la porta per poter riconoscere il protocollo associato, nel tempo, con la diffusione di un numero sempre maggiore di applicazioni, questo metodo è stato superato della tecninca del Deep Packet Inspection.
L'analisi del *payload* di un pacchetto ha reso la classificazione di rete molto più precisa, in modo da poter suddividere il traffico anche a livello applicativo (ad esempio `Facebook` rispetto a `Skype` o `Whatsapp`). Grazie a specfici *dissector*, la classificazione si è potuta spingere sempre più ad alto livello, oltre a migliorare il riconoscimento dei protocolli nei livelli inferiori.

Con il DPI però non abbiamo alcuna percezione del tipo di dato che sta viaggiando in rete. La crescita dei dati multimediali ha posto il problema di riuscire a capire che tipologia di traffico si muova, in modo da **caratterizzare** il traffico da un punto di vista diverso: per protocolli generici come HTTP c'è bisogno anche di caratterizzare il tipo di dato.
Con il DPI non possiamo valutare se quel traffico sia conforme alle politiche interne di una rete.

Da questa considerazione è nata l'idea di introdurre il Machine Learning per il riconoscimento automatizzato delle immagini estratte da traffico HTTP. Inseriamo un ulteriore livello di classificazione, introducendo un modello di ML in grado di riconoscere le immagini secondo due categorie scelte: *lecite* o *illecite*.

Questo ha permesso la creazione di un prototipo per un *framework* di caratterizzazione dei conteenuti di rete.
La libreria nDPI è utilizzata per l'estrazione dell'immagine e dell'*url* sorgente, in modo che, passato il dato al modello creato, quest'ultimo classifichi il contenuto secondo una delle categorie sopra citate. Se la classe riconosciuta è quella *illecita*, il risultato della rete neurale verrà utilizzato dalla libreria di DPI per inserire l'*url*, associato all'immagine, in una *blacklist*.

Il lavoro più dispendioso è dovuto all'implementazione della rete neurale

per creare il modello di classificazione automatica delle immagini. I risultati ottenuti sono soddisfacenti, tenendo in considerazione il fatto che per ottenere valori più alti di accuratezza sarebbe necessario uno studio più approfondito che vada oltre le ore di lavoro di questo tirocinio. Ad esempio è possibile migliorare il riconoscimento valutando altri parametri in base anche alla qualità del dataset che abbiamo.
L'algoritmo utilizzato, **AlexNet**, risulta comunque una delle scelte migliori per problemi legati al riconoscimento delle immagini. Ciò non toglie che sia doveroso anche testare altri algoritmi in modo tale da poter complementare il lavoro e rendere l'efficienza ancora più alta.

7.0.1 Lavori futuri

Il metodo proposto pone la base per lo sviluppo di molti scenari che possano portare benefici a seconda del tipo di contenuto che vogliamo identificare all'interno del traffico di rete. Per prima cosa è possibile estendere il modello con ulteriori classi di riconoscimento (ad esempio *violenza*) in modo da personalizzare la caratterizzazione a seconda delle esigenze di chi utilizza *software* di monitoraggio o filtraggio.
Se si avesse la necessità di aggiungere molte più categorie, sarebbe possibile anche sviluppare un'altra rete neurale che lavori in parallelo alla prima, in modo che i due modelli complementino il lavoro e garantiscano un'accuratezza più elevata, guardando caratteristiche diverse dell'immagine (utilizzo di filtri per rendere l'immagine in bianco e nero o in bassorilievo in modo da risaltare i contorni, ad esempio).

Dato il crescente numero di protocolli cifrati, è possibile pensare anche a scenari in cui si riesca a catalogare il contenuto del traffico anche se criptato. Prendiamo ad esempio una LAN molto grande: implementando il *framework* sviluppato nel tirocinio, è possibile, grazie ad uno o più *transparent proxy*, che il traffico criptato sia decifrato e il contenuto multimediale passato al modello di rete neurale, così da identificare e bloccare sorgenti non consone alla politica interna alla rete.

Questa può essere l'estenzione di un *software* per il filtraggio dei contenuti inappropriati, adottabile nelle scuole o in luoghi in cui la rete sia utilizzata da molti utenti non associabili ad un indirizzo IP. E' impensabile infatti di avere una lista completa di siti o *server* da dover bloccare a priori; è necessario intraprendere questa strada alternative che si muova in parallelo alle tecninche già esistenti.

Bibliografia

[1] S. S. Krishnan and R. K. Sitaraman, *Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-Experimental Designs*, Vol. 21, no. 6, pp. 2001-2014, Dec. 2013.

[2] R. Abbas, *Human perceived quality-of-service for multimedia applications*,
2012 International Conference on Multimedia Computing and Systems, Tangier, 2012, pp. 1-6.

[3] `https://it.wikipedia.org/wiki/Qualit\%C3\%A0_di_servizio#Meccanismi_di_QoS_in_Internet`
Wikimedia Foundation 2017.

[4] `http://www.augiero.it`
Giuseppe Augiero, *Sicurezza delle reti*, 2007.

[5] `http://www.ntop.org/products/traffic-analysis/ntop`
High-Speed Web-based Traffic Analysis and Flow Collection.

[6] `http://www.ieee802.org/1/pages/802.1x-rev.html`
802.1X-REV - Port Based Network Access Control.

[7] `http://www.projectpact.eu/privacy-security-research-paper-series`
Implication of DPI for private security.

[8] R. Alshammari and A. N. Zincir-Heywood,
Machine learning based encrypted traffic classification: Identifying SSH and Skype,
2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications, Ottawa, ON, 2009, pp. 1-8.

[9] Taimur Bakhshi and Bogdan Ghita, *On Internet Traffic Classification: A Two-Phased Machine Learning Approach*
Journal of Computer Networks and Communications, vol. 2016, Article ID 2048302, 21 pages, 2016.

[10] `http://www-users.cs.umn.edu/~yjin/papers/proposal.pdf`
Yu Jin, *Tackling Network Managment problems using ML techniques.*

[11] `http://netgroup.polito.it/teaching/trc/0708/payload-based\%20traffic\%20classification.pdf`
Fulvio Risso, *Application Layer CLassification.*

[12] `https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml`
IANA Functions - Application for Service Names and User Port Numbers.

[13] Patrick Schneider, *TCP/IP Classification on Port Number*
Division of Applied Sciences, Cambridge, MA 2138 (1996).

[14] A. Finamore, M. Mellia, M. Meo and D. Rossi, *KISS: Stochastic Packet Inspection Classifier for UDP Traffic*,
IEEE/ACM Transactions on Networking, vol. 18, no. 5, pp. 1505-1515, Oct. 2010.

[15] Bossert, G., Guihéry, F. and Hiet, G., *Towards automated protocol reverse engineering using semantic information.*
In Proceedings of the 9th ACM symposium on Information, computer and communications security (pp. 51-62). ACM.

[16] `http://www.wireshark.org`, Wireshark Foundation.

[17] Bujlow, Tomasz, Valentín Carela-Español, and Pere Barlet-Ros,
Extended independent comparison of popular deep packet inspection (DPI) Tools for Traffic Classification.
Universitat Politècnica de Catalunya, 2014.

[18] `http://github.com/wanduow/libprotoident`
libprotoident 2.0.10, *Network traffic classification library*,
Copyright © 2011-2016 The University of Waikato, Hamilton, New Zealand.

[19] `http://netfilter.org`
Netfliter project, Copyright © 1999-2014 Harald Welte, Pablo Neira Ayuso.

[20] `http://l7-filter.clearos.com`
Copyright © ClearFoundation 2009-2017.

[21] Hjelmvik, Erik, and Wolfgang John. *Statistical protocol identification with spid*,
Swedish National Computer Networking Workshop. 2009.

[22] `https://arxiv.org/abs/1511.08899`
Applying deep learning to classify pornographic images and videos,
Mohamed N. Moustafa, Department of Computer Science and Engineering, The American University in Cairo.

[23] Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada

[24] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. CoRR, abs/1409.4842 (2014)

[25] `http://www.ntop.org/products/deep-packet-inspection/ndpi`, Open and Extensible LGPLv3 Deep Packet Inspection Library.

[26] Bujlow, Tomasz, Valentín Carela-Español, and Pere Barlet-Ros. *Independent comparison of popular DPI tools for traffic classification.* Computer Networks 76 (2015): 75-89.

[27] Deri, Luca, et al. *ndpi: Open-source high-speed deep packet inspection*, Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International. IEEE, 2014.

[28] `http://github.com/betolj/ndpi-netfilter`, GPL implementation of an iptables and netfilter module for nDPI integration into the Linux kernel.

[29] `http://multifast.sourceforge.net`, Multifast: an open-source software for an efficient implementation of the Aho-Corasick algorithm as a C library.

[30] `http://en.wikipedia.org/wiki/Trie`, Trie data srtucture

[31] `http://www.chromium.org/spdy/spdy-whitepaper`, SPDY: An experimental protocol for a faster web.

[32] `http://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html`

[33] `https://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html`

[34] An Introduction to Statistical Learning, Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani, © Springer Science+Business Media New York 2013

[35] `http://bias.csr.unibo.it/maltoni/ml/DispensePDF/DeepLearning.pdf` Davide Maltoni, Università di Bologna, (*Deep Learning*)

[36] Yann LeCun official site - `http://yann.lecun.com`

[37] https://sites.google.com/site/pornographydatabase
NPDI Pornography Database, (2013), Brazil.

[38] https://www.tensorflow.org/
An open-source software library for Machine Intelligence

[39] API Tensorflow - https://www.tensorflow.org/api_docs/

[40] Pickle module - https://docs.python.org/2/library/pickle.html

[41] Adam optimizer - https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer

[42] TensorBoard - https://www.tensorflow.org/get_started/summaries_and_tensorboard

[43] Peafowl - https://github.com/DanieleDeSensi/Peafowl