PYNQports

Utilizzo di Reti Neurali Convoluzionali su FPGA per Riconoscimento di Caratteri Falsificati nei Passaporti

Report di Progetto - Sistemi Digitali M
A.A. 2023/2024

Riccardo Bovinelli, Francesca Bocconcelli, Lorenzo Riccardi

14 Febbraio 2024

Indice

Capitolo 1

Introduzione

1.1 Scopo del progetto

Perno centrale di questo progetto è stato lo sviluppo di un *tool* a riga di comando che, ricevuta in input una lista di immagini di passaporti, potesse rilevare certe categorie di caratteri falsificati [1] tramite una rete neurale convoluzionale (Convolutional Neural Network, CNN) implementata su FPGA e che producesse un output indicante l'area dell'immagine in cui sono presenti falsificazioni. La board scelta per implementare la CNN è una PYNQ - Z2 di Xilinx.

1.2 PYNQ - Z2

La PYNQ - Z2 è una board che ha come obiettivo principale la semplificazione dell'interfacciamento con piattaforme Xilinx, in questo caso la Zynq 7000. Essa è dotata di un processore ARM (interno al Processing System, PS) e di una FPGA (Programmable Logic, PL), il cui interfacciamento è facilitato dal meccanismo degli *Overlay*, ovvero un'interfaccia che incapsula componenti FPGA e li rende utilizzabili lato Python. In altre parole, essi servono per demandare l'esecuzione ad un HW diverso da quello del processore. Tramite queste entità si carica un *bitstream* (file che contiene la configurazione dell'hardware) sulla FPGA della board. Le comunicazioni tra PL e PS sono effettivamente gestite da un componente chiamato *Driver*. L'ambiente PYNQ mette a disposizione driver di default che permettono di interfacciarsi con qualsiasi *IP Core* (blocco di componenti logici a sè stante) caricato sulla PL.

Capitolo 2

Svolgimento

Il progetto si è sviluppato in maniera orizzontale, e tutti i partecipanti del gruppo hanno preso parte contemporaneamente a tutte le fasi del processo. La prima fase è stata la creazione del dataset, prendendo spunto da [1]. Successivamente si è proceduto a creare ed eseguire il training di una CNN che potesse riconoscere i caratteri reali e quelli falsificati. Al raggiungimento di risultati soddisfacenti, si è generato il relativo *bitstream* (e l'overlay per ambiente PYNQ) con l'utilizzo della libreria hls4ml [2]. Infine si è eseguito il deployment sulla board, il relativo testing e si è realizzata un'interfaccia a riga di comando che permette di sfruttare la CNN caricata sulla FPGA. Seguono le fasi di progettazione nel dettaglio.

2.1 Creazione dataset

Il progetto sviluppato ha richiesto un'ampia mole di dati, necessari ad addestrare una Convolutional Neural Network (CNN) che riconoscesse caratteri falsificati all'interno di passaporti, distinguendo passaporti veri da passaporti falsi. Non è stato possibile riferirsi ad un dataset preesistente, poiché non esisteva un insieme di dati specifico contenente immagini adatte allo scopo. Di conseguenza, si è ricorsi alla generazione di un dataset personalizzato, come sarà puntualmente descritto a breve.
L'articolo [1] ha fornito le fondamenta per la metodologia adottata nella creazione del dataset.

Inizialmente, si sono acquisite immagini di passaporti da fonti ufficiali [1] in modo da ottenere un riferimento di base per le immagini contenute nel dataset. In seguito, si sono creati più file CSV contenenti informazioni anagrafiche

[1] `https://www.consilium.europa.eu/prado`

fittizie (nome, cognome, sesso, data e luogo di nascita), che hanno il compito di fornire elementi testuali da incorporare nel dataset, mantenendo comunque un certo grado di plausibilità nei dettagli del set di dati.

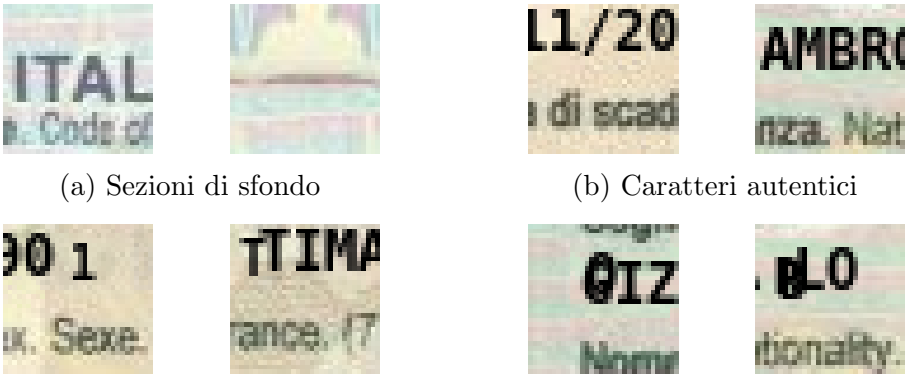
Per trasformare questi dati in input utilizzabili per la CNN, è stato sviluppato un codice in linguaggio Python in grado di leggere i dati dai file CSV e creare frame 32x32 contenenti diversi tipi di caratteri falsificati in svariate posizioni. La scelta della dimensione è stata frutto di un bilanciamento tra quantità di informazione contenuta in un'immagine ed efficienza di elaborazione su una board PYNQ, che ha ridotte capacità di calcolo.
Durante questo processo, si sono identificati i passaporti falsi come quelli contenenti elementi fuorvianti, come caratteri leggermente distorti o sovrapposti. Si è stabilita la tipologia di carattere falso da rilevare e si sono generati alcuni passaporti con tali caratteristiche in posizioni casuali e in dimensioni variabili.
In fase di generazione dei passaporti fittizi (figura 2.1), le coordinate contenenti questi caratteri sono state memorizzate e le immagini suddivise in modo da ottenere i singoli frame del dataset (figura 2.2).

Figura 2.1: Esempio di passaporto con caratteri falsificati

Poiché l'obiettivo finale della rete era permettere di distinguere tra passaporti autentici e falsificati, si è dovuto categorizzare ogni tipologia di frame contenuta nei passaporti (background, informazioni lecite, caratteri falsificati) affinché potesse effettuare questa distinzione. Sono stati quindi necessarie tre distinte funzioni generative: una per la creazione di caratteri falsi, un'altra per la generazione di frame di sfondo ed una terza per la creazione di caratteri autentici.

Durante lo sviluppo del dataset, si è dedicata una quantità consistente di tempo nella scelta degli offset dei caratteri all'interno delle immagini per garantire un apprendimento coerente da parte della rete. Fattori come gli offset sui caratteri e la presenza o assenza di caratteri parzialmente visibili influenzano infatti significativamente il processo di addestramento.

Nel complesso si sono identificate quattro classi di elementi per l'addestramento della CNN: immagini di sfondo, immagini con caratteri genuini, immagini con caratteri sfalsati e immagini con caratteri sovrapposti. Il numero di elementi effettivamente generati per classe è stato, rispettivamente: 936 per le immagini di sfondo, 15 746 per i caratteri reali, 7 388 per i caratteri sfalsati e 6 442 per i caratteri sovrapposti.



(a) Sezioni di sfondo

(b) Caratteri autentici

(c) Caratteri sfalsati

(d) Caratteri sovrapposti

Figura 2.2: Esempio di immagini 32x32 contenute nel dataset

Una volta create le immagini relative a ciascuna classe, tramite il framework Tensorflow, è stato possibile creare un'entità dataset per training della rete. In particolare Tensorflow fornisce un metodo, facente parte della libreria Keras, che permette di caricare un dataset di immagini a partire da una directory locale.

```
ds_train = tf.keras.utils.image_dataset_from_directory(
      data_dir,
      validation_split=0.2,
      label_mode="categorical",
      subset="training",
      seed=123,
      image_size=(img_height, img_width),
      batch_size=batch_size
)
```

Listing 2.1: Training della rete

Con `image_dataset_from_directory()` si crea un dataset di dati di training a partire dalla directory locale "data_dir", con il parametro `validation_split` si indica alla libreria di utilizzare il 20% degli elementi presenti nella directory per la fase di validation della rete, al termine di ogni epoca. Oltre a quest'informazione, è stata specificata la dimensione dei batch da fornire alla rete, ovvero il numero di immagini che vengono processate insieme durante l'addestramento del modello. Il parametro `label_mode` impostato a "categorical", ha lo scopo di assegnare etichette in un preciso formato alle classi presenti nei dati. Nel dettaglio, le etichette vengono codificate come vettori *one-hot*, in cui ogni classe è rappresentata da una singola cella di un vettore a valori binari (di lunghezza pari al numero delle classi del dataset, `n_classes`). Ad ogni immagine viene quindi assegnato un vettore di lunghezza `n_classes` con una singola cella contenente il valore 1 nella posizione corrispondente alla classe d'appartenenza e valore 0 in tutte le altre posizioni. Questo tipo di codifica è comunemente utilizzato nelle reti neurali per la classificazione multiclasse, in cui ogni immagine appartiene ad una delle almeno 3 classi di destinazione.
Con lo stesso metodo è stato creato il dataset di validation, ovvero un insieme separato di dati utilizzato per valutare le prestazioni del modello durante l'addestramento. Per questo motivo, questi dati non vengono utilizzati per aggiornare i pesi del modello, ma vengono utilizzati per monitorare l'andamento delle metriche di valutazione (come l'accuracy o la loss) su un set di dati indipendente.

Infine, a tutte le immagini del dataset è stata applicata un'operazione di preprocessing, in particolare un'operazione di *normalizzazione* dei valori `RGB` delle immagini, per rendere più uniforme ed efficiente l'addestramento della rete. Dopo questa normalizzazione, le immagini di input sono array quadridimensionali di dati (`np.float32`) compresi tra lo 0 e l'1, rendendo così più limitato il range di differenze tra di loro.

2.2 Progettazione della rete neurale

La fase successiva del progetto è stata quella della scelta della struttura della rete neurale convoluzionale da implementare. Dato che la strategia attuata per il riconoscimento di caratteri falsificati ricade nella sequenziale *image classification* di sezioni 32x32 dell'immagine di input, si è scelto di utilizzare una rete neurale convoluzionale adatta allo scopo. La struttura scelta, e descritta di seguito, è stata trovata in rete ed adattata alle nostre necessità (figura 2.3).

La struttura del modello relativa alla CNN scelta è composta da layer convoluzionali alternati a layer di pooling, seguiti da uno o più layer densi e uno strato di output softmax (`tf.keras.layers.Softmax`).

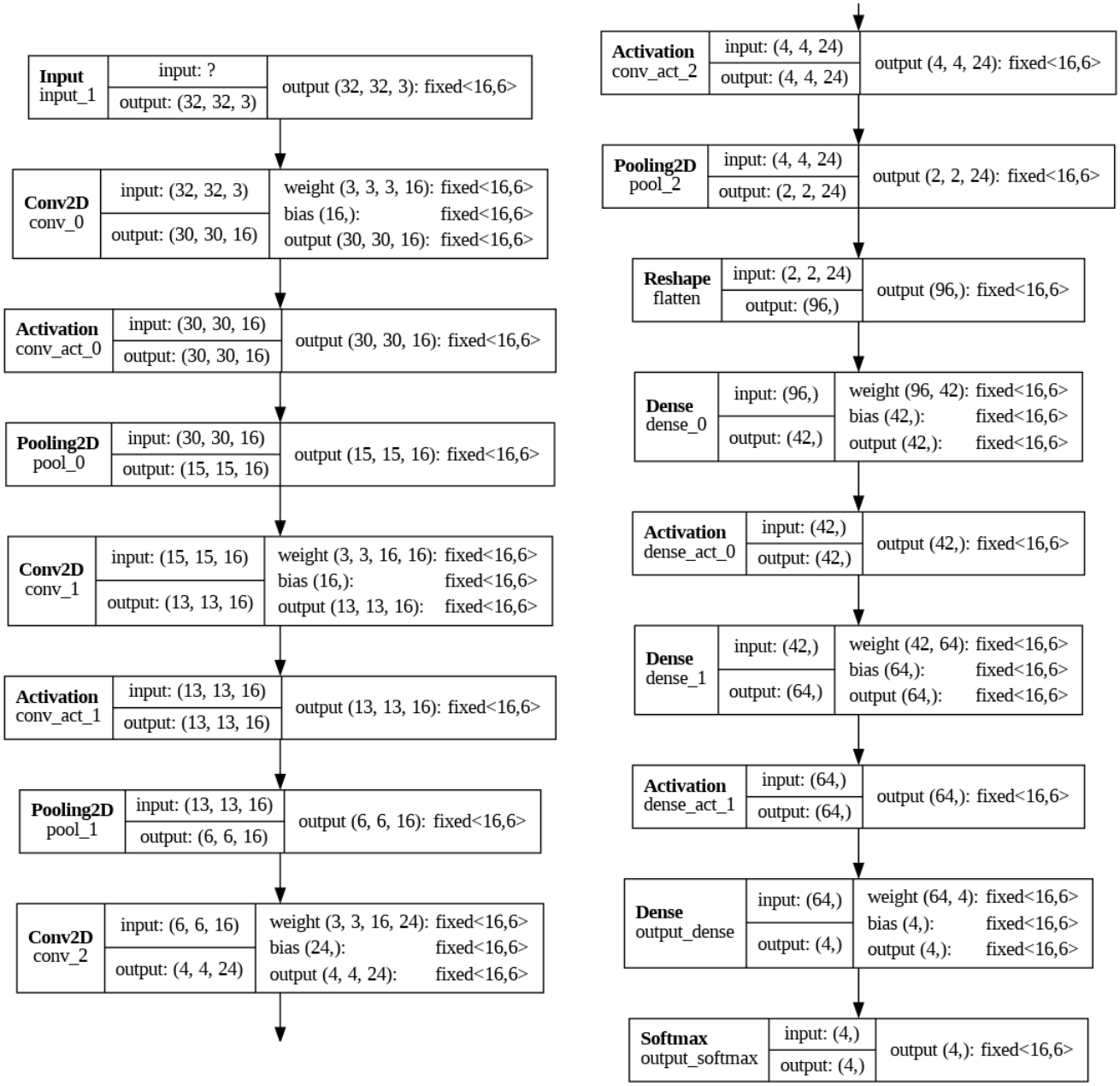


Figura 2.3: Schema della struttura della rete (ottenuto tramite [2])

I dati di training, o dati di input, vengono forniti alla rete attraverso un livello di input, che è il primo livello della rete. Successivamente, i dati passano attraverso gli strati convoluzionali, che sono responsabili dell'estrazione delle caratteristiche significative dalle immagini in input, anche definite *feature map*. Le caratteristiche in questione potrebbero essere, ad esempio, le lettere presenti dentro a una singola immagine di training.
Ogni blocco convoluzionale in una CNN è composto da diversi componenti chiave. Uno di questi è il layer `tf.keras.layers.Conv2D`, che applica una serie di filtri di convoluzione alla feature map di input. Questi filtri aiutano a individuare caratteristiche distintive all'interno dell'immagine.
Dopo l'applicazione dei filtri di convoluzione, viene eseguita una normalizzazione dei batch utilizzando `tf.keras.layers.BatchNormalization`. Questo aiuta a stabilizzare e accelerare l'addestramento della rete, garantendo che i dati in input siano normalizzati e quindi più facilmente ottimizzabili. È stato inserito questo layer nonostante sia già stata fatta un'operazione di preprocessing sulle immagini per rendere massima la normalizzazione dei dati.
Successivamente, viene applicata una funzione di attivazione (`tf.keras.layers.Activation`) ReLU (Rectified Linear Unit). Questa funzione introduce non linearità nella rete, aiutandola a imparare pattern complessi e non lineari presenti nei dati di input.
Dopo ogni blocco convoluzionale, viene inserito uno strato di pooling (`tf.keras.layers.MaxPooling2D`), ovvero una tecnica utilizzata per ridurre la dimensione della feature map, mantenendo solo le informazioni più rilevanti. Il MaxPooling2D prende il valore massimo all'interno di una finestra di pooling e lo utilizza come valore rappresentativo per quella regione della feature map, contribuendo così a ridurre la dimensione spaziale dei dati e a rendere la rete più efficiente computazionalmente.
Dopo l'elaborazione attraverso gli strati convoluzionali, la feature map risultante viene "appiattita" in un vettore unidimensionale tramite uno strato di flatten. Questo processo è necessario per passare dalla rappresentazione spaziale della feature map a una forma lineare, permettendole di essere processata da uno o più strati densi. Il layer Flatten di TensorFlow, o `tf.keras.layers.Flatten`, esegue questa operazione di appiattimento, trasformando la feature map in un vettore unidimensionale senza alterare il contenuto delle informazioni.
Successivamente, questo vettore viene passato attraverso uno o più strati densi, rappresentati da `tf.keras.layers.Dense`. Ogni strato denso è composto da un numero specificato di neuroni, i quali sono collegati a tutti i neuroni dello strato precedente. Questo permette alla rete di apprendere relazioni complesse tra le caratteristiche estratte durante le fasi precedenti.
Ogni strato denso è generalmente seguito poi da una normalizzazione dei

batch e da una funzione di attivazione ReLU.
Infine, lo strato di output consiste in un altro strato denso con un numero di neuroni pari al numero di classi nel problema di classificazione. L'output di questo strato è passato attraverso una funzione di attivazione softmax, che converte i valori di output in probabilità, fornendo la distribuzione di probabilità delle diverse classi per ciascun campione di input. Questo permette di interpretare l'output della rete come le probabilità predette per ciascuna classe, facilitando così la fase di classificazione.

Dal momento che lo scopo ultimo del progetto è quello di inserire il modello all'interno della Board PYNQ, passando dalla generazione del bitstream tramite Vivado, si è scelto di effettuare un'operazione di *pruning aware training* dei livelli di convoluzione e densi mirando ad un valore di sparsità di circa il 50%. Questa operazione permette di ridurre la complessità del modello, eliminando le connessioni meno importanti o meno utilizzate. Questo viene fatto forzando alcuni pesi del modello a zero, rendendo così le corrispondenti matrici dei pesi *sparse* (matrici in cui quasi tutti i valori sono pari a zero).
Questa operazione è di notevole vantaggio quando si implementano modelli di deep learning su FPGA, dispositivi che, di fatto, hanno risorse limitate. Ridurre la complessità del modello tramite il pruning può aiutare a garantire che il modello possa adattarsi alle risorse disponibili su un dispositivo FPGA e permettere di avere maggiore efficienza al prezzo di una piccola perdita di accuratezza.

2.3 Training e build della CNN con Vivado

In questa sezione si tratterà dell'argomento centrale del progetto, ovvero la fase di training del modello Keras e la conseguente conversione del modello in bitstream e generazione dell'overlay per l'ambiente PYNQ.
Grazie alla libreria *hls4ml* [2] è stato possibile convertire un modello di rete neurale realizzato con il framework Tensorflow in un flusso di dati di descrizione hardware compatibile con le esigenze del progetto.

2.3.1 hls4ml

La libreria hls4ml è una libreria di alto livello che fornisce vari servizi orientati alla realizzazione di reti neurali su FPGA, tra cui la generazione di *bitstream* a partire da modelli Keras (libreria per reti neurali che si appoggia su Tensorflow) e *overlays* compatibili con l'ambiente PYNQ.

Astraendo dalla logica sottostante, questa libreria offre la possibilità di trasformare direttamente un modello di rete realizzato con framework quali

Figura 2.4: Logo del progetto hls4ml di FastML

Tensorflow o PyTorch in un progetto HLS, del quale è poi possibile, sempre attraverso chiamate a funzione fornite, eseguirne la compilazione interfacciandosi con ambienti di progettazione già disponibili nel sistema (nel nostro caso Xilinx Vivado 2019.1).

2.3.2 Training della CNN

Pruning dei pesi

La tecnica di pruning dei pesi è una strategia che permette di comprimere una rete neurale e, di conseguenza, di ridurre l'utilizzo delle risorse. Nello specifico, la strategia adottata, denominata **magnitude-based pruning**, implica l'eliminazione dei pesi ridondanti nei tensori, ponendo a zero i pesi di valore minore. Tutte le moltiplicazioni con pesi nulli vengono omesse dalla libreria HLS durante la traduzione della rete in firmware, risparmiando quindi significative risorse FPGA.
Il pruning viene applicato utilizzando l'API di pruning di TensorFlow, un'interfaccia basata su Keras composta da un semplice sostituto plug-and-play dei layer di Keras. Si mira a ottenere una sparsity del 50%, il che significa che solo il 50% dei pesi viene mantenuto nel layer pruned e gli altri vengono impostati a zero.
Prima del pruning, i pesi di ogni layer vengono inizializzati ai pesi del modello corrispondente senza pruning, garantendo che il modello sia in un minimo stabile prima di rimuovere i pesi ritenuti non importanti.
A ogni modello viene applicato il pruning a partire dalla decima epoca, con sparsity target che aumenta gradualmente fino al 50%.

```
def pruneFunction(layer):
    pruning_params = {
        'pruning_schedule': sparsity.PolynomialDecay(
            initial_sparsity=0.0, final_sparsity=0.50,
   begin_step=NSTEPS * 2, step=NSTEPS * 10, frequency=NSTEPS
        )
```

```
    }
    if isinstance(layer, tf.keras.layers.Conv2D):
        return tfmot.sparsity.keras.prune_low_magnitude(layer
    , **pruning_params)
    if isinstance(layer, tf.keras.layers.Dense) and layer.
    name != 'output_dense':
        return tfmot.sparsity.keras.prune_low_magnitude(layer
    , **pruning_params)
    return layer
```

Listing 2.2: Pruning della rete

L'addestramento viene eseguito minimizzando la categorical crossentropy loss utilizzando l'ottimizzatore Adam. La categorical crossentropy loss è una funzione di perdita comunemente utilizzata nell'addestramento di modelli di classificazione multiclasse. Questa funzione va a misurare la discrepanza tra le probabilità previste dal modello e le vere etichette di classe dei dati di addestramento. L'obiettivo è quello di, durante l'addestramento, ridurre questa perdita, cioè rendere le previsioni del modello il più vicine possibile alle etichette reali.
L'ottimizzatore Adam è un algoritmo di ottimizzazione ampiamente utilizzato nell'addestramento dei modelli di deep learning, che combina i vantaggi di due popolari algoritmi di ottimizzazione, AdaGrad e RMSProp, per aggiornare in modo efficiente i pesi del modello durante l'addestramento. Nello specifico, Adam utilizza un tasso di apprendimento adattativo per ciascun parametro, che viene aggiornato in modo dinamico durante il processo di addestramento. Questo adattamento aiuta ad accelerare la convergenza durante l'addestramento e può evitare problemi come l'instabilità del gradiente.
Il tasso di apprendimento ottimale viene impostato inzialmente a 0.003. Se non vi è alcun miglioramento nella perdita per cinque epoche, il tasso di apprendimento viene ridotto del 90% fino a raggiungere un tasso di apprendimento minimo di 10^{-6}.

```
LOSS = tf.keras.losses.CategoricalCrossentropy()
OPTIMIZER = tf.keras.optimizers.Adam(learning_rate=3e-3,
    beta_1=0.9, beta_2=0.999, epsilon=1e-07, amsgrad=True)
```

Listing 2.3: Perdita e ottimizzazione

Per evitare overfitting, viene attivato l'arresto anticipato (`tf.keras.callbacks.EarlyStopping`) quando non viene osservato alcun miglioramento nella funzione di perdita per dieci epoche consecutive.

```
callbacks = [
    tf.keras.callbacks.EarlyStopping(patience=10, verbose=1),
    tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
    factor=0.5, patience=3, verbose=1),
```

```
    pruning_callbacks.UpdatePruningStep(),
]
```

Listing 2.4: Callbacks

Il numero di epoche impiegato per il training è di 30, train_data è il dataset di training ottenuto come descritto nel paragrafo 2.1. In più, per comprendere il carico computazionale di questo addestramento, è stato aggiunto un codice di controllo del tempo impiegato a svolgere il fitting del modello.

```
start = time.time()
history = model_pruned.fit(train_data, epochs=n_epochs,
    validation_data=val_data, callbacks=callbacks)
end = time.time()
```

Listing 2.5: Training della rete

Tenendo traccia degli andamenti dell'accuracy (valutata sui dati di training) e della val_accuracy (accuratezza valutata sull'insieme di validation) è possibile costruire un grafico che ne rappresenta l'andamento in funzione delle epoche di training trascorse.

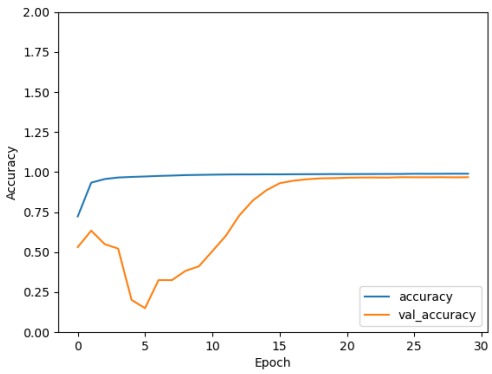


Figura 2.5: Plot dell'accuratezza del modello

Inoltre, di seguito è riportato un plotting dei valori che assumono i pesi all'interno dei livelli della rete neurale. È possibile notare come il *pruning aware training* abbia un impatto significativo, infatti è molto evidente la presenza di pesi con valore 0.

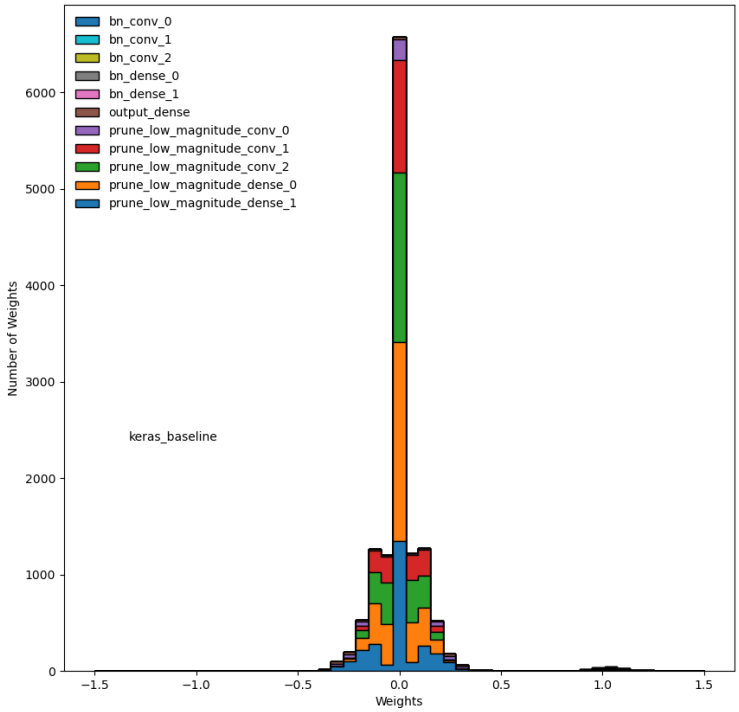


Figura 2.6: Plot della distribuzione dei valori dei pesi

2.3.3 Generazione del bitstream

A seguito del training della rete, ottenendo un modello `tf.keras.Model` si è proceduto alla conversione in un bitstream.
Per effettuare questa conversione utilizzando hls4ml si deve derivare un file di configurazione (che verrà utilizzato per passare delle direttive a Vivado) dal modello di rete scelto.

```
1 hls_config = hls4ml.utils.config_from_keras_model(model,
    granularity='name')
```

Listing 2.6: Generazione del file di configurazione

Dato che il bitstream doveva essere caricato su PYNQ, che non ha un dispositivo FPGA di grandi dimensioni, è stato necessario aggiungere delle direttive che permettessero di poter ottenere una descrizione hardware adeguata, come aumentare il `ReuseFactor` (ovvero un meccanismo della libreria hls4ml che serve a specificare quante volte devono essere riutilizzati i moltiplicatori

sul dispositivo, se viene settato a 1 c'è parallelismo massimo) ad un valore consono. Tramite *trial-and-error* si sono trovati soddisfacenti questi valori di riutilizzo per ogni livello:

```
hls_config['LayerName']['conv_0']['ReuseFactor'] = 108
hls_config['LayerName']['conv_1']['ReuseFactor'] = 144
hls_config['LayerName']['conv_2']['ReuseFactor'] = 144
hls_config['LayerName']['dense_0']['ReuseFactor'] = 96
hls_config['LayerName']['dense_1']['ReuseFactor'] = 84
hls_config['LayerName']['output_dense']['ReuseFactor'] = 128
```

Listing 2.7: Fattori di riutilizzo per ogni layer

Per creare un bitstream ad hoc per la board PYNQ è necessario un apposito backend denominato "VivadoAccelerator", successivamente si può procedere alla compilazione del progetto hls tramite `hls_model.compile()`.

```
cfg = hls4ml.converters.create_config(backend='
    VivadoAccelerator')
    ...
# configurazione del backend VivadoAccelerator
    ...
hls_model = hls4ml.converters.keras_to_hls(cfg)
hls_model.compile()
```

Listing 2.8: Creazione del modello HLS

Se la compilazione ha successo, con la chiamata a funzione `hls_model.build()` si può passare ad eseguire il building del bitstream e alla generazione opzionale di un overlay per gestire i componenti FPGA.

```
hls_model.build(export=True, bitfile=True)
```

Listing 2.9: Building del modello HLS e generazione del bitfile e dell'overlay

Una volta terminata la compilazione, analizzando il log di Vivado, si possono rilevare le percentuali di utilizzazione dei componenti hardware all'interno della FPGA.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	36222	0	53200	68.09
LUT as Logic	36698	0	53200	63.34
LUT as Memory	2524	0	17400	14.51
LUT as Distributed RAM	22	0		
LUT as Shift Register	2502	0		
Slice Registers	52575	0	106400	49.41
Register as Flip Flop	52575	0	106400	49.41
Register as Latch	0	0	106400	00.00
F7 Muxes	1266	0	26600	4.76
F8 Muxes	417	0	13300	3.14

2.4 Interfaccia a linea di comando

Al fine di utilizzare il software sviluppato in questo progetto è stato creato uno script in linguaggio Python, con interfaccia a riga di comando.
Si è quindi scelta la seguente interfaccia:

```
$ python pynqports.py img1 ... imgN
```

con img1 ... imgN un elenco di path che si riferiscono ad immagini con formato supportato da NumPy.

Lo script farà eseguire l'inferenza sulla rete neurale convoluzionale per ciascuna immagine in ingresso, separandole in singoli frame: il risultato di questa operazione sarà quindi l'insieme di tutti i frame 32x32, analizzati uno ad uno con le relative probabilità di appartenenza ad una data classe. Quando un frame risulta appartenere ad una classe corrispondente ad un carattere errato (in qualche misura) lo si etichetta con una bounding box (rossa o gialla, a seconda del margine di incertezza sulla misurazione) intorno al frame di appartenenza: un esempio di questa procedura è mostrato nella sezione 2.5. Infine tutti i frame verranno utilizzati per ricostruire l'immagine finale, che verrà salvata in un file, nella stessa directory sorgente, denominato `imgK_analyzed.png`.

Scendendo più nel dettaglio, per la forte necessità di ottimizzazione data dalla bassa potenza di calcolo del processore ARM montato sulla board PYNQ, si sono svolte più operazioni di refactoring del codice, in modo da limitare l'utilizzo delle risorse di calcolo presenti nella PYNQ ed ottimizzare i tempi di risposta dalla rete. In particolare, la libreria NumPy ha permesso di velocizzare lo svolgimento di alcuni calcoli che, eseguiti con le funzionalità offerte dalla libreria standard Python, sarebbero risultati decisamente più onerosi (un esempio è l'overloading dell'operatore * utilizzato alla fine per riportare i float32 nell'intevallo [0, 255]). La logica di analisi dei frame e di ricomposizione delle immagini è quindi la seguente:

```
    i = 0
    output_file = path.join(img_output_folder, "".join(
file_arg.split(".")[:-1]) + "_analyzed.png")
    reconstructed_image = np.ndarray((416,0,3))

    for classification in predictions_1:
        max_class = np.max(classification)
        if (max_class == classification[2] or max_class ==
classification[3]):
            slices[i,:,:,:] = addBBox(slices[i,:,:,:],
max_class)

```

```
10          if i % int((img_height / 32)) == 0:
11              if i != 0:
12                  reconstructed_image = np.concatenate((
    reconstructed_image, column), axis=1)
13              column = slices[i,:,:,:]
14          else:
15              column = np.concatenate((column, slices[i,:,:,:])
    , axis=0)
16          i += 1
17
18      Image.fromarray((reconstructed_image * 255).astype(np.
    uint8)).save(output_file)
```

Listing 2.10: Snippet dello strumento CLI

La filosofia seguita è quella di successive concatenazioni dei frame: prima in una singola colonna, poi aggiungendo la colonna alle altre precedentemente costruite.

2.5 Risultati ottenuti

In questa sezione si mostrerà un esempio di risultati ottenuti dall'elaborazione di un passaporto falsificato. Passata come argomento di input al tool a riga di comando l'immagine `fake_passport_1.png` (figura 2.7)

Figura 2.7: `fake_passport_1.png`, input della rete neurale convoluzionale

si ottiene in uscita `fake_passport_1_analyzed.png` (figura 2.8).

Figura 2.8: `fake_passport_1_analyzed.png`, output del tool

Si nota come i caratteri falsificati siano stati individuati dalla rete neurale che gira su FPGA e le sezioni del passaporto che li contengono siano state contornate di rosso. In questo caso è stato rilevato un carattere sovrapposto in alto a sinistra, nel cognome, ed un carattere sfalsato in basso a sinistra nell'area della data di scadenza.

Capitolo 3

Conclusioni

In questo progetto si è analizzata la fattibilità di un deployment di rete neurale convoluzionale sulla parte FPGA della scheda PYNQ-Z2.

In particolare, la rete neurale ottenuta e utilizzata per il riconoscimento di caratteri falsi in passaporti, sebbene gli errori ricercati appartenessero a classi ben specifiche, si è rivelata sufficientemente efficiente e precisa: oltre ai vantaggi in efficienza forniti da una implementazione su hardware ad-hoc, tra cui basso consumo energetico ed alta velocità possibile grazie all'elevato grado di parallelismo nelle operazioni, si è riscontrata una soddisfacente accuratezza nell'individuazione delle falsificazioni.
La rete ottenuta non è chiaramente perfetta in quanto saltuariamente presenta falsi negativi però, in compenso, raramente mostra falsi positivi ed è sufficientemente affidabile per discernere passaporti veri da passaporti con caratteri falsificati.

Bibliografia

[1] Teerath Kumar et al. «Forged Character Detection Datasets: Passports, Driving Licences and Visa Stickers». In: *International Journal of Artificial Intelligence & Applications* 13 (mar. 2022), pp. 21–35. DOI: 10.5121/ijaia.2022.13202.

[2] FastML Team. *fastmachinelearning/hls4ml*. Ver. v0.8.1. 2023. DOI: 10.5281/zenodo.1201549. URL: https://github.com/fastmachinelearning/hls4ml.