

Progetto di Riconoscimento Facciale

Ghidini Matteo

3 febbraio 2026

Indice

1	Introduzione	3
2	Architettura Generale	4
2.1	Flusso Logico del Sistema	4
3	Analisi dei Moduli Core	5
3.1	detector.py	5
3.2	recognizer.py	5
3.3	utils.py	6
3.4	initializer.py	7
3.4.1	Metodo create_model	8
4	Preparazione Dati ed Entrypoint	11
4.1	Script di Preparazione immagini	11
4.2	Script Eseguibili (Entrypoint)	12
5	Flow di utilizzo applicazione	16
5.1	Utilizzo immediato	16
5.2	Aggiunta di persona al dataset di riconoscimento	16
5.3	Aggiunta di persona al dataset per operazione Look-alike	16
6	Interfaccia Grafica (GUI)	17
6.1	Struttura e Funzionalità Principali	17
6.2	Flusso di Lavoro: Aggiunta di una Nuova Persona	18
6.3	Esecuzione degli Script Principali	18
7	Reti Neurali dell'applicazione	20
7.1	Modello Pre-addestrato: ResNet per Face Recognition	21
7.1.1	Descrizione	21
7.1.2	Motivazione	21
7.1.3	Implementazione	21
7.1.4	Dettagli Tecnici	21
7.2	Deep Neural Network (DNN)	22
7.2.1	Descrizione	22
7.2.2	Motivazione	22
7.2.3	Implementazione	22
7.2.4	Dettagli Tecnici	23
7.2.5	Configurazioni Testate	23
7.3	Convolutional Neural Network (CNN)	26
7.3.1	Descrizione	26

7.3.2	Motivazione	26
7.3.3	Implementazione	26
7.3.4	Dettagli Tecnici	30
7.3.5	Architettura CNN classica (use_residual=false)	31
7.3.6	CNN con Resudual Blocks (use_residual=true)	32
7.3.7	ResidualBlock Struttura Interna	33
7.3.8	Configurazioni Testate	33
7.4	Transfer Learning e Fine-Tuning	38
7.4.1	Descrizione	38
7.4.2	Motivazione	38
7.4.3	Implementazione	38
7.4.4	Dettagli Tecnici	40
7.4.5	Configurazioni Testate	40
7.5	Training delle architetture	45
8	Confronto Finale tra le Architetture	51

Sommario

Questa relazione presenta una descrizione dell'applicazione sviluppata per il progetto di Deep Learning e Architetture di Reti Neurali. La relazione è divisa in diverse sezioni:

- [sezione 1](#) descrive il progetto e i suoi obiettivi
- [sezione 2](#) illustra il flusso operativo dell'applicazione
- [sezione 3](#) presenta gli snippet di codice più rilevanti e le componenti fondamentali dell'applicazione
- [sezione 4](#) descrive il processo di elaborazione dei dati (immagini) per l'utilizzo nell'applicazione
- [sezione 5](#) spiega come utilizzare l'applicazione tramite interfaccia a riga di comando
- [sezione 6](#) illustra l'utilizzo della GUI dell'applicazione
- [sezione 7](#) descrive le reti neurali utilizzate, la creazione con i Colab Notebooks, la loro implementazione e l'adattamento al contesto applicativo
- [sezione 8](#) presenta i risultati del confronto tra le reti

1 Introduzione

L'applicazione sviluppata consente due principali operazioni: riconoscimento del viso delle persone e confronto del viso delle persone con altre (look-alike search)

Idea del funzionamento: invece di avere una Rete neurale che classifica, ho rimosso i Logit Layer (e.g: last fully connected layer) e ho come risultato un vettore che rappresenta in uno spazio n dimensionale l'immagine di input. Idealmente vettori vicini significano facce simili, quindi calcolando la distanza tra vettori posso identificare e confrontare i visi delle persone. Si basa su due parti principali:

- **detector:** dato un frame ritaglia i bounding box dei visi
- **recognizer:** dato un viso fa l'embedding e lo confronta con quelli presenti nei dataset.

L'applicazione è un'unione di diversi script python che interagiscono tra loro.

Sono presenti 4 Main, rispettivamente:

- **main_camera:** utilizzo della videocamera per prendere le immagini live e riconoscerle, confrontandole con le immagini nel dataset
- **main_image:** riconoscimento dei visi nelle immagini
- **main_look_alike_online:** utilizzo della videocamera per prendere le immagini live e riportare i visi che assomigliano maggiormente ad esso
- **main_look_alike_offline:** confrontare le immagini nel dataset con i visi e trovare quelli che assomigliano maggiormente ad esso

I file `main_*.py` possono essere utilizzati direttamente da Bash ma è caldamente consigliato l'utilizzo della GUI data la mole di comandi e argomenti da inserire.

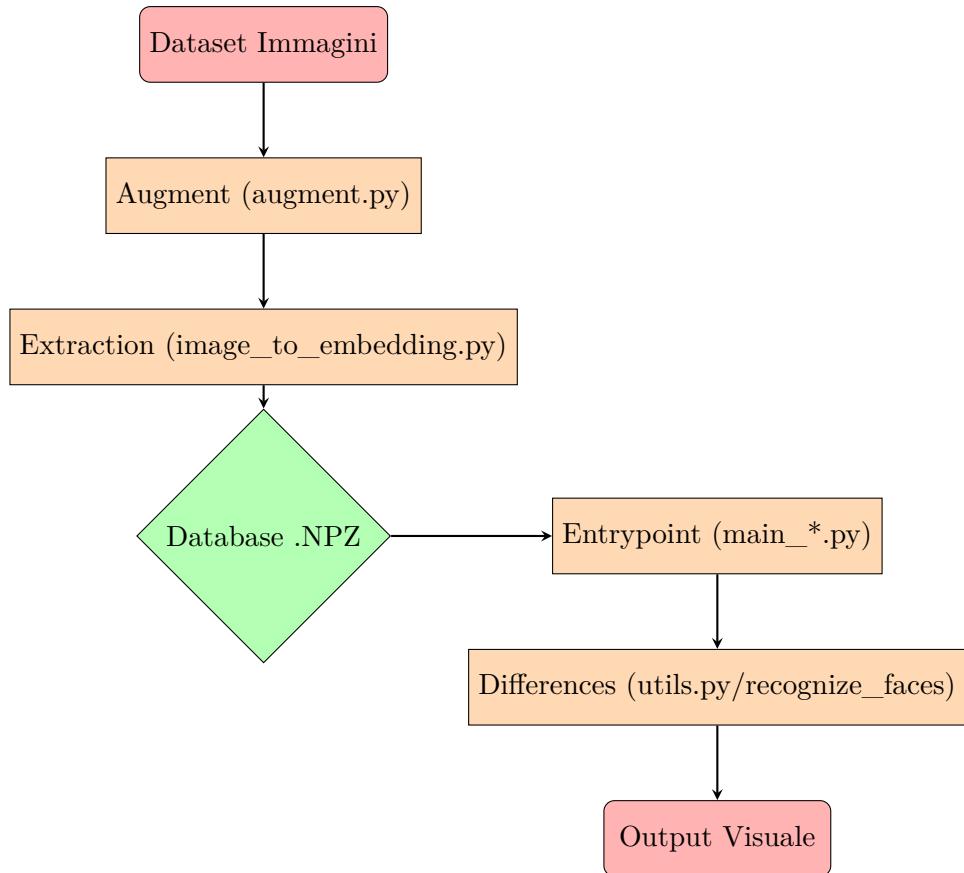
2 Architettura Generale

L'applicazione è suddivisa in tre aree logiche: componenti Core, script di preparazione e entry-point esecutivi.

La configurazione è centralizzata per garantire modularità e manutenibilità in un file config.py, dove si possono impostare cartelle di output degli embedding e cartelle dove sono presenti i pesi dei modelli (di detector e recognizer)

2.1 Flusso Logico del Sistema

Il seguente diagramma illustra come i dati fluiscono dal dataset grezzo fino all'output di riconoscimento in tempo reale.



3 Analisi dei Moduli Core

I componenti Core forniscono le funzioni base necessarie all'intero ecosistema.

3.1 detector.py

Contiene la classe `FaceDetector` per l'individuazione dei volti tramite YOLO. Il metodo `detect` astrae l'inferenza del modello e restituisce un elenco di bounding box e score di confidenza.

```

1 def detect(self, frame):
2     """
3         Rileva volti in un frame BGR.
4         Returns:
5             dictionary con:
6                 List[Tuple[int, int, int, int]]: lista di bounding box (x1, y1,
7                 x2, y2)
8                 score: confidenza della rilevazione
9             """
10    results = self.model(frame, conf=self.conf, verbose=False)
11    faces = []
12
13    for result in results:
14        if not result.boxes:
15            continue
16
17        boxes = result.boxes.xyxy.cpu().numpy()
18        scores = result.boxes.conf.cpu().numpy()
19
20        for i in range(len(boxes)):
21            x1, y1, x2, y2 = map(int, boxes[i])
22            faces.append({
23                'bbox': (x1, y1, x2, y2),
24                'score': float(scores[i])
25            })
26
27    return faces

```

Listing 1: Metodo detect in FaceDetector

Relazioni: Fornisce l'input (crop del volto) ai moduli di riconoscimento.

3.2 recognizer.py

Implementa `FaceRecognizer` per la generazione di embedding vettoriali. Il metodo `get_embedding` riceve un'immagine ritagliata di un volto, la preprocessa e la passa al modello (rete neurale) per ottenere un vettore normalizzato.

```

1 def get_embedding(self, face_bgr: np.ndarray) -> np.ndarray:
2     """
3         Estraie embedding normalizzato L2.
4     """
5     face_tensor = self._preprocess(face_bgr)
6
7     with torch.no_grad():
8         if self.use_get_embedding:
9             embedding = self.model.get_embedding(face_tensor)
10        else:
11            embedding = self.model(face_tensor)

```

```

13     embedding = embedding.cpu().numpy()[0]
14
15     # Normalizzazione L2
16     norm = np.linalg.norm(embedding)
17     embedding = embedding / norm if norm > 0 else embedding
18
19     return embedding

```

Listing 2: Metodo get_embedding in FaceRecognizer

Relazioni: Trasforma i dati visuali in dati numerici per i confronti di similarità.

Se il modello presenta il metodo get_embeddings significa che è una classe creata da me. Posso utilizzare direttamente questo che mi ritorna il vettore senza logit layer. Se invece è un modello esterno la classe FaceRecognizer rimuove i logit layers dal forward

```

1     state_dict = {
2         k: v for k, v in state_dict.items()
3         if not k.startswith("logits.") and not k.startswith("classifier."))
4     }
5
6     # Carica i pesi nel modello (strict=False per ignorare layer
7     # mancanti come classifier)
8     self.model.load_state_dict(state_dict, strict=False)

```

3.3 utils.py

Collezione di funzioni helper. La funzione `recognize_faces` orchestra il processo di riconoscimento: rileva i volti, calcola l'embedding per ciascuno e lo confronta con il database pre-caricato.

```

1 def recognize_faces(frame, detector, recognizer, embeddings_array,
2                     labels_list, threshold=0.60):
3     """
4     Rileva volti, calcola embedding, confronta con dataset.
5     """
6     faces = detector.detect(frame)
7     results = []
8
9     for face in faces:
10         (x1, y1, x2, y2) = face['bbox']
11         face_crop = frame[y1:y2, x1:x2]
12         name = "Unknown"
13         confidence = 0.0
14
15         emb = recognizer.get_embedding(face_crop)
16
17         if embeddings_array.size > 0:
18             # cosine similarity = dot product (embedding L2-normalizzato
19             # con array)
20             sims = embeddings_array @ emb
21             best_idx = np.argmax(sims)
22             best_sim = sims[best_idx]
23
24             if best_sim > threshold:
25                 name = labels_list[best_idx]
26                 confidence = (best_sim - threshold) / (1.0 - threshold)

```

```

26     results.append({
27         "bbox": (x1, y1, x2, y2),
28         "name": name,
29         "confidence": confidence
30     })
31
32 return results

```

Listing 3: Funzione recognize_faces in utils.py

Relazioni: Utilizzato dagli entrypoint per orchestrare le fasi di detection e recognition.

3.4 initializer.py

Script per inizializzare i modelli.

I modelli di recognizer serve crearli attraverso una factory che utilizza le classi python dei modello (contenute in src/model_recognition/) e le configurazioni nei file dei pesi (contenuti in models/face_recognition/*.pth, contenete sia i pesi che la configurazione usata per la rete specifica)

```

1 # Crea il modello usando la factory
2 if model_config is None:
3     # Carica config dal checkpoint (se disponibile)
4     model_info = create_model(
5         backbone_type=backbone_type,
6         checkpoint_path=recognizer_model_path
7     )
8 else:
9     # Usa config fornita
10    model_info = create_model(
11        backbone_type=backbone_type,
12        checkpoint_path=recognizer_model_path,
13        **model_config
14    )
15
16 # Crea il recognizer con tutti i parametri necessari
17 recognizer = FaceRecognizer(
18     model=model_info['model'],
19     model_path=recognizer_model_path,
20     image_size=model_info['image_size'],
21     embedding_size=model_info['embedding_size'],
22     use_get_embedding=model_info['use_get_embedding']
23 )
24
25 # Crea il modello usando la factory
26 if model_config is None:
27     # Carica config dal checkpoint (se disponibile)
28     model_info = create_model(
29         backbone_type=backbone_type,
30         checkpoint_path=recognizer_model_path
31     )
32 else:
33     # Usa config fornita
34     model_info = create_model(
35         backbone_type=backbone_type,
36         checkpoint_path=recognizer_model_path,
37         **model_config
38     )

```

```

39     # Crea il recognizer con tutti i parametri necessari
40     recognizer = FaceRecognizer(
41         model=model_info['model'],
42         model_path=recognizer_model_path,
43         image_size=model_info['image_size'],
44         embedding_size=model_info['embedding_size'],
45         use_get_embedding=model_info['use_get_embedding']
46     )
47

```

3.4.1 Metodo create_model

Metodo factory utilizzato nell'initializer per gestire le diverse reti possibili

```

1 def create_model(backbone_type, checkpoint_path=None, **model_kwargs):
2     """
3         Factory per creare modelli di face recognition.
4
5     Args:
6         backbone_type: "InceptionResnetV1" o "FaceEmbeddingCNN"
7         checkpoint_path: path al .pth/.pt (opzionale, per caricare
config da li)
8             **model_kwargs: parametri per costruire il modello (override
della config del checkpoint)
9
10    Returns:
11        dict con:
12            - 'model': istanza del modello PyTorch
13            - 'image_size': dimensione input attesa (es. 160 o 128)
14            - 'embedding_size': dimensione embedding output (es. 512,
128, 256)
15            - 'use_get_embedding': True se il modello ha get_embedding()
, False se usa forward()
16
17    Examples:
18        # InceptionResnetV1
19        model_info = create_model("InceptionResnetV1")
20
21        # FaceEmbeddingCNN con config da checkpoint
22        model_info = create_model("FaceEmbeddingCNN", checkpoint_path="models/my_model.pth")
23
24        # FaceEmbeddingCNN con config manuale
25        model_info = create_model(
26            "FaceEmbeddingCNN",
27            checkpoint_path="models/my_model.pth",
28            num_filters=[64, 128, 256],
29            embedding_size=256
30        )
31    """
32
33    if backbone_type == "InceptionResnetV1":
34        model = InceptionResnetV1(pretrained=None)
35        return {
36            'model': model,
37            'image_size': 160,
38            'embedding_size': 512,

```

```

39         'use_get_embedding': False # InceptionResnetV1 usa forward
40     ())
41
42     elif backbone_type == "FaceEmbeddingCNN":
43         # Se trova checkpoint, prova a caricare la config da lì
44         config = {}
45
46         if checkpoint_path:
47             try:
48                 checkpoint = torch.load(checkpoint_path, map_location='cpu')
49                 if isinstance(checkpoint, dict) and 'config' in checkpoint:
50                     checkpoint_config = checkpoint['config']
51
52                     # Se la config ha model\config annidato, estrailo
53                     if 'model_config' in checkpoint_config:
54                         config = checkpoint_config['model_config'].copy()
55
56                 else:
57                     # Altrimenti usa direttamente config (compatibilità)
58                     config = checkpoint_config.copy()
59
60             except Exception as e:
61                 print(
62                     f" Warning: impossibile caricare config da
63 checkpoint: {e}")
64
65             # Override con parametri forniti da utente (hanno priorità massima)
66             config.update(model_kwargs)
67
68             # Valori di default SOLO se config è completamente vuota
69             if not config:
70                 config = {
71                     'input_channels': 3,
72                     'num_filters': [32, 64, 128],
73                     'kernel_sizes': [3, 3, 3],
74                     'fc_hidden_size': 512,
75                     'embedding_size': 128,
76                     'dropout_rate': 0.5,
77                     'use_batchnorm': True,
78                     'use_global_avg_pool': False,
79                     'use_residual': False
80                 }
81             print("Warning: nessuna config trovata, uso valori di
82 default")
83
84             # Crea il modello
85             model = create_cnn_model(config)
86
87             return {
88                 'model': model,
89                 'image_size': 128, # Le tue CNN usano 128x128
90                 'embedding_size': config.get('embedding_size', 128),
91                 'use_get_embedding': True
92             }

```

```
89         # FaceEmbeddingCNN ha get_embedding()
90     }
91
92     #... continua per anche la classe della DNN e del Transfer
Learning e Fine Tuning
```

Relazioni: utilizzato dai quattro main e dal test_model per ricevere facilmente modello di recognizer e detector

4 Preparazione Dati ed Entrypoint

Le immagini sono inserite nelle apposite cartelle, manualmente o via GUI, e poi si estraggono gli embedding dei visi.

Vi sono diverse cartelle contenente immagini e i loro embeddings che sono gestite dalla gui, ma è possibile l'inserimento manuale (sconsigliato dato che bisogna seguire uno schema preciso di nominazione delle cartelle):

- **data**: contiene tutti i dati dell'applicazione, immagini di persone, embeddings, metadati.
- **data/dataset**: contiene cartelle contenenti ognuna le immagini di una persona e gli embedding di queste.
Queste immagini sono la base per tutti i main_*.py, dato che forniscono gli embeddings con cui fare i confronti
- **data/classify_images**: immagini da classificare per il main_image.py.
- **data/similarity_images/known_people**: immagini utilizzate per il main_look_alike_offline
- **data/similarity_images/people**: immagini utilizzate per il main_look_alike_*; si confrontano le immagini in data/similarity_images/known_people oppure dalla telecamera, con queste immagini.
- **data/test_images**: contiene cartelle, ognuna col nome della persona e contenete immagini di essa; queste vengono usate per fare testing delle reti neurali grazie a test_models.py

È presente un controllo di versione via hash: gli embeddings delle immagini sono computati solo se l'hash delle immagini delle specifiche cartelle è diverso da quello salvato nei metadati. questo permette di non recomputare inutilmente immagini.

È importante notare che per ogni modello che si utilizza, gli embedding saranno diversi, quindi un nuovo file .npz verrà creato

4.1 Script di Preparazione immagini

augment.py : codice utilizzato per fare augmentazione delle immagini, modifiche con rotazioni, crop, light-changes, flip, per avere immagini più variate.

```

1 def random_flip(img):
2     if random.random() < 0.5:
3         return cv2.flip(img, 1)
4     return img
5
6
7 def random_brightness(img):
8     alpha = random.uniform(0.7, 1.3)
9     beta = random.randint(-20, 20)
10    return cv2.convertScaleAbs(img, alpha=alpha, beta=beta)
11
12
13 def random_crop(img, crop_ratio=0.9):
14     h, w = img.shape[:2]
15     ch, cw = int(h * crop_ratio), int(w * crop_ratio)
16     y = random.randint(0, h - ch)
17     x = random.randint(0, w - cw)
18     cropped = img[y:y+ch, x:x+cw]
19     return cv2.resize(cropped, (w, h))
20

```

```

21
22 def augment_image(img):
23     img = random_flip(img)
24     img = random_brightness(img)
25     img = random_crop(img)
26     return img

```

Relazioni: Script autonomo ma le immagini augmentate sono embeddeddate dai modelli.

image_to_embedding.py: Pipeline che utilizza *detector* e *recognizer* per generare i database vettoriali da inserire in **data/dataset**: Itera sulle immagini, rileva i volti, calcola gli embedding e li salva in formato .npz.

```

1 for img_path, img_type in image_files:
2     fname = os.path.basename(img_path)
3     img = cv2.imread(img_path)
4     if img is None:
5         metadata["skipped_images"].append(fname)
6         continue
7
8     faces = detector.detect(img)
9     if len(faces) != 1:
10        print(f"WARN {person_name}/{fname}: {len(faces)} faces, skip")
11        metadata["skipped_images"].append(fname)
12        continue
13
14     x1, y1, x2, y2 = faces[0]["bbox"]
15     face_crop = img[y1:y2, x1:x2]
16
17     emb = recognizer.get_embedding(face_crop)
18     embeddings[fname] = emb

```

Listing 4: Ciclo principale in image_to_embedding.py

extract_embedding.py: Simile a image_to_embebbing.py ma per le immagini di **data/similarity_images/***.

4.2 Script Eseguibili (Entrypoint)

main_camera.py : Sistema di riconoscimento in tempo reale. Il ciclo principale cattura i frame dalla webcam e invoca **recognize_faces**.

```

1 while True:
2     ret, frame = cap.read()
3     if not ret:
4         break
5
6     results = recognize_faces(
7         frame, detector, recognizer, embeddings_array, labels_list)
8
9     # Disegna bounding box
10    for r in results:
11        x1, y1, x2, y2 = r["bbox"]
12        name = r["name"]
13        confidence = r["confidence"]
14        draw_label(frame, name, confidence, (x1, y1, x2, y2))

```

```
15 cv2.imshow("Face Recognition", frame)
16 if cv2.waitKey(1) & 0xFF == 27: # ESC per uscire
17     break
18
```

Listing 5: Ciclo principale in main_camera.py

main_image : Sistema di riconoscimento visi in immagini inserite nella cartella **data/classify_images**. Ad esempio nella [Figura 1](#)

main_look_alike_online & main_look_alike_offline : Sistema di ritrovamento di visi simili a quelli forniti, sia via camera che via immagini presenti nella cartella **data/similarity_images/people**. Ad esempio nella [Figura 2](#)

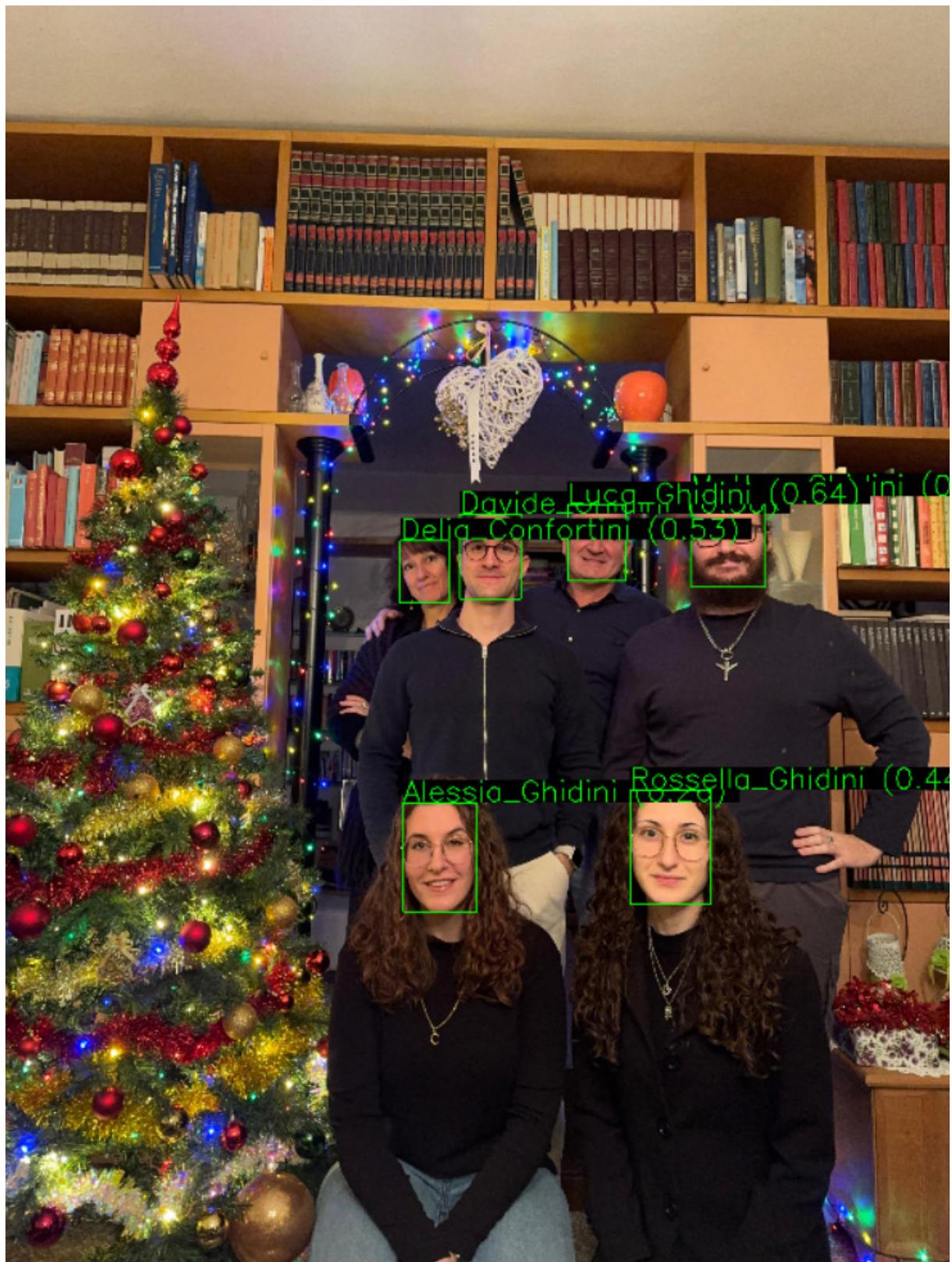


Figura 1: Famiglia Ghidini classificata correttamente

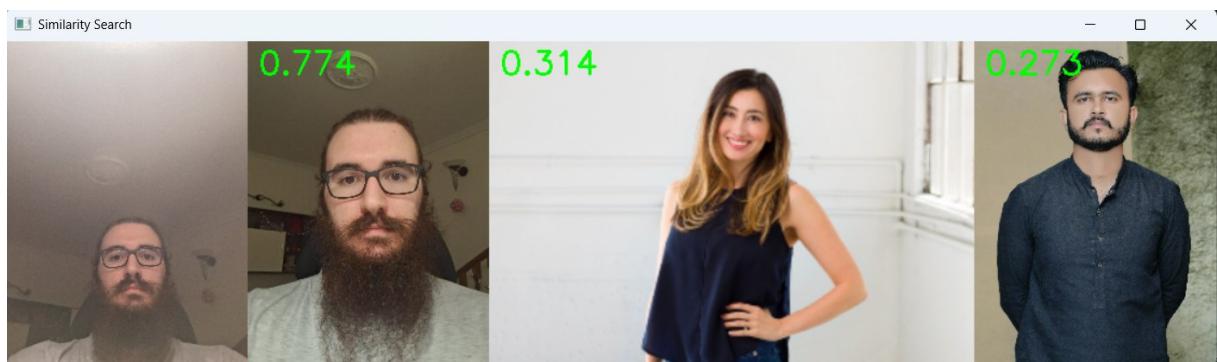


Figura 2: A sinistra immagine di Matteo Ghidini nel sistema Look alike, a destra 3 immagini di persone e la somiglianza con esse

5 Flow di utilizzo applicazione

5.1 Utilizzo immediato

Se si vuole far semplicemente partire un'applicazione basta scrivere su :

```
python -m src.main_*
```

Questo farà partire una dei main presenti in src utilizzando le configurazioni standard (Modello ResNet-image) se non si specificano parametri.

5.2 Aggiunta di persona al dataset di riconoscimento

Se invece si vuole aggiungere persone alla knowledge base serve fare alcuni passaggi aggiuntivi:

1. Aggiungere alla dir `data/dataset` la dir con nome quello della persona che si vuole aggiungere alla KB
2. Eseguire su bash shell:
`python -m augment`
 Questo augmenterà l'immagini presenti nel dataset
3. Per creare gli embeddings dalle immagini: `python -m src.image_to_embedding`
4. Si può far partire i `main_camera` o `main_image` con
`python -m src.main_*`

5.3 Aggiunta di persona al dataset per operazione Look-alike

Se invece si vuole aggiungere persone nel dataset dei Look-alike serve fare alcuni passaggi aggiuntivi e il primo punto cambia per modalità online (via camera) e offline (img nel dataset)

1. Aggiungere alla dir `data/similarity_images/known_people` l'immagine con nome quello della persona che si vuole aggiungere alla lista di persone che si volgono confrontare con i Look-alike
2. Aggiungere alla dir `data/similarity_images/people` le immagini di persone "sconosciute" con cui si vuole fare un confronto (nota che questo passaggio è abbastanza computational intensive data la mole delle immagini). Aggiungere e quindi ricreare gli embeddings della dir `similarity_images/people` solo se disposti ad aspettare del tempo per la ricomputazione)
3. Per creare gli embeddings dalle immagini in `similarity_images`:
`python -m src.extract_embeddings`
 Nota che di base è commentata la parte di codice che fa l'embeddings delle `similarity_images/people` per sicurezza (ultime righe del file .py)
4. Si può far partire i main interessati al look-alike comparison con
`python -m src.main_look_alike_offline` per confrontare immagini in `similarity_images/known_people`;
`python -m src.main_look_alike_online` per confrontare direttamente attraverso la camera.

6 Interfaccia Grafica (GUI)

Per semplificare l'interazione con il sistema, è stata sviluppata un'applicazione grafica `gui_app.py` basata su Tkinter. La GUI astrae l'uso degli script da riga di comando, guidando l'utente attraverso le operazioni più comuni.

Il comando per far partire la gui:

```
python -m src.gui_app
```

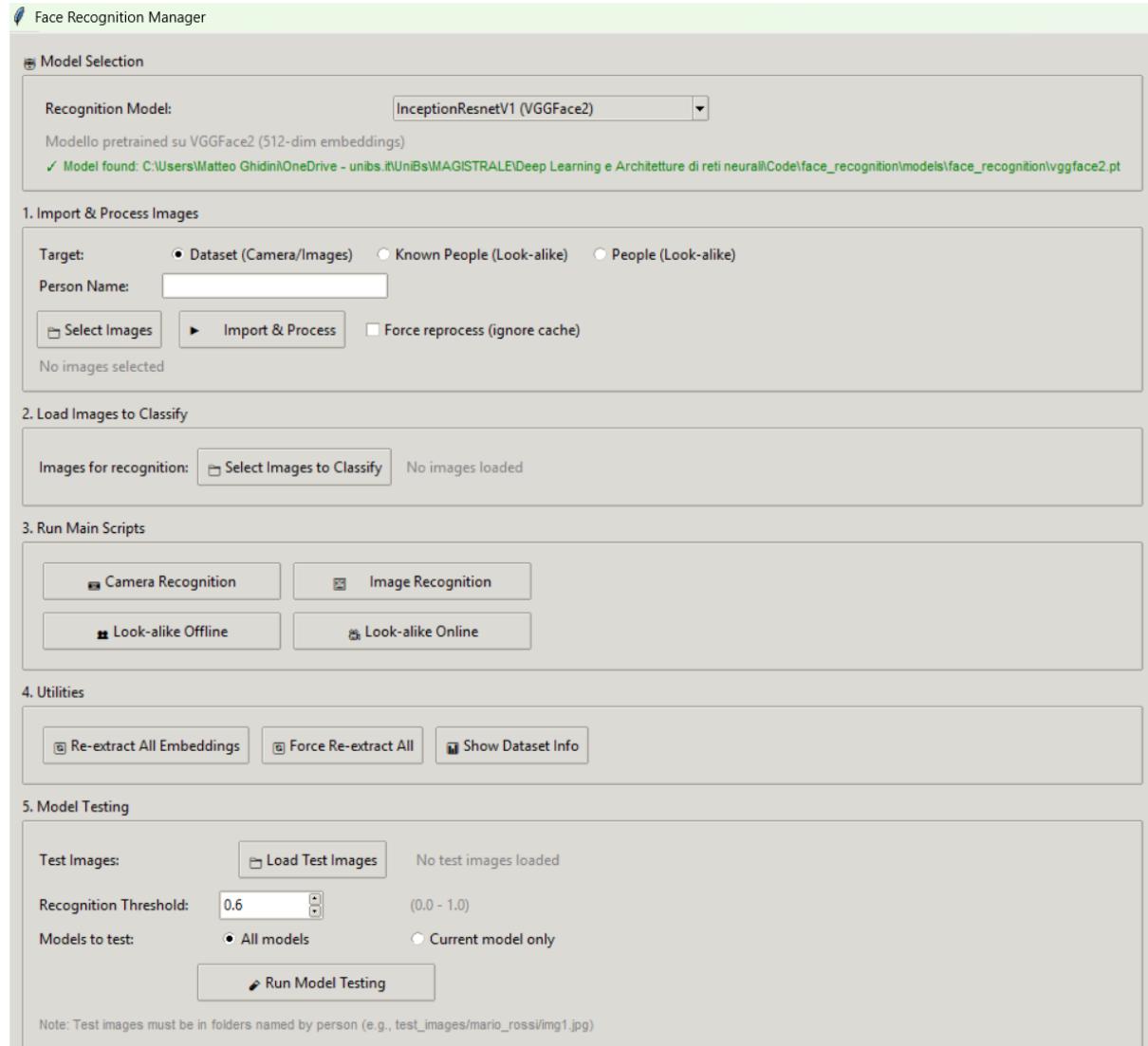


Figura 3: Panoramica dell'interfaccia grafica.

6.1 Struttura e Funzionalità Principali

L'interfaccia è suddivisa in sezioni numerate che rappresentano un flusso di lavoro logico:

1. **Model Selection:** Permette di selezionare il modello di riconoscimento da utilizzare per tutte le operazioni. Mostra una descrizione del modello e verifica l'esistenza dei file dei pesi.
2. **Import & Process Images:** Sezione per aggiungere nuove persone al dataset. L'utente inserisce il nome, seleziona le immagini e avvia un processo che importa i file, esegue l'augmentazione (se necessaria) e computa gli embedding.

3. **Load Images to Classify:** Carica le immagini da usare per il riconoscimento delle immagini da classificare.
4. **Run Main Scripts:** Pulsanti per avviare i quattro script principali dell'applicazione (riconoscimento da camera, da immagine e ricerca look-alike).
5. **Utilities:** Funzioni per la manutenzione del database di embedding, come la ri-estrazione forzata o incrementale e informazioni riguardo agli embeddings presenti nel dataset.
6. **Model Testing:** Una sezione dedicata a eseguire test sistematici sui modelli, calcolando metriche di accuratezza e generando report.
7. **Console Output:** Un'area di testo che mostra i log di tutte le operazioni eseguite, fornendo feedback in tempo reale.

Tutte le operazioni che richiedono tempo (come l'estrazione di embedding o l'esecuzione di script) vengono eseguite in thread separati per non bloccare l'interfaccia.

6.2 Flusso di Lavoro: Aggiunta di una Nuova Persona

L'aggiunta di una nuova persona è gestita dal metodo `_do_import_and_process`. Questo metodo orchestra la copia dei file, l'augmentazione e l'estrazione degli embedding invocando gli script corrispondenti tramite `subprocess`.

```

1 def _do_import_and_process(self, person_name, target, model_name):
2     try:
3         # Determine target directory based on 'target'
4         if target == "dataset":
5             base_dir = Path(DATASET_DIR)
6             person_dir = base_dir / person_name / "images"
7             needs_augment = True
8             script = "image_to_embedding.py"
9             # ... (other targets)
10
11         # Copy images
12         for file in self.selected_files:
13             # ... (copy logic)
14
15         # Run augmentation if needed
16         if needs_augment:
17             self.log("Running augmentation...")
18             subprocess.run([sys.executable, "-m", "src.augment"], ...)
19
20         # Extract embeddings
21         self.log(f"Extracting embeddings with model: {model_name}...")
22         cmd = [sys.executable, "-m", f"src.{script.replace('.py', '')}"]
23         cmd.extend(["--model", model_name])
24         subprocess.run(cmd, ...)
25
26     except Exception as e:
27         self.log(f"Error: {str(e)}", "ERROR")

```

Listing 6: Logica di importazione e processamento in `gui_app.py`

6.3 Esecuzione degli Script Principali

L'esecuzione degli script principali è gestita dal metodo `_do_run_script`, che riceve il nome dello script e il modello selezionato, e li esegue in un sottoprocesso.

```
1 def _do_run_script(self, script_name, model_name):
2     try:
3         # Pass the selected model as an argument
4         result = subprocess.run(
5             [sys.executable, "-m", script_name, "--model", model_name],
6             capture_output=True,
7             text=True
8         )
9         self.log(result.stdout)
10        if result.returncode != 0:
11            self.log(f"{script_name} failed: {result.stderr}", "ERROR")
12    except Exception as e:
13        self.log(f"Error running {script_name}: {str(e)}", "ERROR")
```

Listing 7: Esecuzione di uno script principale in gui_app.py

7 Reti Neurali dell'applicazione

Nell'applicazione sono state implementate e testate diverse architetture di reti neurali, ciascuna con obiettivi specifici. Le reti utilizzate spaziano da modelli pre-addestrati state-of-the-art a implementazioni custom da zero, permettendo un confronto diretto tra approcci di diversa complessità e dimostrando l'evoluzione delle prestazioni in base all'architettura scelta.

I risultati delle CNN e le CNN allenate con Transfer Learning e Fine Tuning sono riportate due volte dato che sono stati utilizzati due dataset:

- LFW:
 - <https://www.kaggle.com/datasets/jessicali9530/lfw-dataset>
 - 13.000 immagini di 5749 persone
 - Utilizzate effettivamente sono le labels che avevano più di 10 immagini
- CelebA:
 - <https://www.kaggle.com/datasets/jessicali9530/celeba-dataset>
 - 200.000 immagini, labellate per caratteristiche
 - utilizzate solo le caratteristiche booleane: male, young, smiling, eyeglasses. Principalmente per non avere caratteristiche sbilanciate (e.g: 5_o_clock_shadow) oppure inutili (e.g: colore dei capelli)

7.1 Modello Pre-addestrato: ResNet per Face Recognition

7.1.1 Descrizione

Per il riconoscimento facciale è stato utilizzato un modello ResNet pre-addestrato, appartenente alla famiglia delle Residual Networks. Questi modelli sono caratterizzati dall'uso di connessioni residuali (skip connections) che permettono di addestrare reti molto profonde evitando il problema del vanishing gradient.

7.1.2 Motivazione

La scelta di utilizzare un modello pre-addestrato per il face recognition è motivata da:

- **Prestazioni elevate:** i modelli ResNet sono stati state-of-the-art per il riconoscimento per task di computer vision.
- **Baseline di riferimento:** fornisce un punto di confronto ottimale per valutare le prestazioni dei modelli custom
- **Efficienza:** evita la necessità di addestrare da zero un modello complesso su dataset di grandi dimensioni
- **Robustezza:** addestrato su milioni di immagini, garantisce eccellente generalizzazione

Data la presenza della ResNet architecture negli import non è stato necessario scrivere una classe per ResNet.

7.1.3 Implementazione

```

1 from facenet_pytorch import InceptionResnetV1
2
3 def create_model(backbone_type, checkpoint_path=None, **model_kwargs):
4     """
5         Factory per creare modelli di face recognition.
6     """
7     if backbone_type == "InceptionResnetV1":
8         model = InceptionResnetV1(pretrained=None)
9         return {
10             'model': model,
11             'image_size': 160,
12             'embedding_size': 512,
13             'use_get_embedding': False    # InceptionResnetV1 usa forward
14         }

```

Listing 8: Creazione del modello ResNet nell'inizializer

7.1.4 Dettagli Tecnici

- **Architettura:** InceptionResnetV1
- **Dataset di pre-training:** VGGFace2, CASIA-WebFace
- **Dimensione embedding:** 512 feature vector
- **Metrica di similarità:** Cosine similarity

7.2 Deep Neural Network (DNN)

7.2.1 Descrizione

È stata implementata una Deep Neural Network completamente connessa (fully connected), composta da multipli layer densi con funzioni di attivazione non lineari. Questa architettura rappresenta l'approccio più basilare al deep learning.

7.2.2 Motivazione

La DNN è stata implementata per:

- **Dimostrazione didattica:** mostrare come le architetture semplici abbiano performance limitate su task complessi
- **Baseline di confronto:** stabilire una baseline minima di prestazioni
- **Comprendere dei limiti:** evidenziare i problemi delle reti fully-connected su dati ad alta dimensionalità

7.2.3 Implementazione

```

1 class FaceEmbeddingDNN(nn.Module):
2     """
3         Deep Neural Network per generare embeddings di volti.
4         Durante training: classificazione delle identità (cross entropy)
5         Durante inference: estrazione embedding dal penultimo layer
6     """
7
8     def __init__(self,
9                  input_size=128*128*3, # immagini 128x128 RGB flattened
10                 hidden_sizes=[1024, 512], # dimensioni layer nascosti
11                 embedding_size=128, # dimensione embedding finale
12                 num_classes=5749, # numero identità in DataSet LFW
13                 dropout_rate=0.5, # dropout rate
14                 use_batchnorm=True): # usa batch normalization
15             """
16             Args:
17                 input_size: dimensione input (img_height * img_width *
18                         channels)
19                         hidden_sizes: lista con dimensioni dei layer nascosti
20                         embedding_size: dimensione del vettore embedding
21                         num_classes: numero di identità (classi) per
22                             classificazione
23                         dropout_rate: probabilità dropout (0 = no dropout)
24                         use_batchnorm: se True, aggiunge BatchNorm dopo ogni layer
25             """
26             super(FaceEmbeddingDNN, self).__init__()
27
28             self.input_size = input_size
29             self.embedding_size = embedding_size
30             self.use_batchnorm = use_batchnorm
31
32             # Costruzione layers sequenziali per feature extraction
33             layers = []
34             prev_size = input_size
35
36             # Hidden layers

```

```

35     for hidden_size in hidden_sizes:
36         layers.append(nn.Linear(prev_size, hidden_size))
37         if use_batchnorm:
38             layers.append(nn.BatchNorm1d(hidden_size))
39         layers.append(nn.ReLU())
40         layers.append(nn.Dropout(dropout_rate))
41         prev_size = hidden_size
42
43     # Embedding layer (penultimo layer - questo e' cio' che useremo
44     # per embeddings)
44     layers.append(nn.Linear(prev_size, embedding_size))
45     if use_batchnorm:
46         layers.append(nn.BatchNorm1d(embedding_size))
47     layers.append(nn.ReLU())
48
49     # Sequential usa i layer che abbiamo appena fatto append. Di
50     # solito sarebbe sequential (layer1, layer2 ...)
51     self.feature_extractor = nn.Sequential(*layers)
52
53     # Classification head (solo per training, rimosso durante
54     # inference)
55     # last alyer fully conencted, da dim ultimo hidden layer a dim
56     # classi
57     self.classifier = nn.Linear(embedding_size, num_classes)
58
59 def forward(self, x):
60     """
61     Forward pass completo per training (con classificazione)
62
63     Args:
64         x: tensor [batch_size, channels, height, width]
65
66     Returns:
67         logits: tensor [batch_size, num_classes]
68     """
69
70     # Flatten dell'immagine
71     x = x.view(x.size(0), -1) # [batch_size, input_size]
72
73     # Estrazione features + embedding
74     embedding = self.feature_extractor(x) # [batch_size,
75     embedding_size]
76
77     # Classificazione
78     logits = self.classifier(embedding) # [batch_size, num_classes]
79
80     return logits

```

Listing 9: Classe DNN

7.2.4 Dettagli Tecnici

Nell'applicazione è utilizzata la migliore DNN trovata, ma diversi modelli sono stati creati per confrontare le prestazioni con la variazione di parametri di configurazione:

Architettura: [Input → Dense(X) → BatchNorm(X) → ReLU → ... → Output]

7.2.5 Configurazioni Testate

Sono state testate diverse architetture per valutare l'impatto del numero di layer e neuroni:

Tabella 1: Configurazioni DNN per Face Recognition

Config	Hidden Sizes	Embedding Size	Dropout	LR	Batch Size	Optimizer
Baseline	[1024, 512]	128	0.5	10^{-3}	64	Adam
No Dropout	[1024, 512]	128	0.0	10^{-3}	64	Adam
No BatchNorm	[1024, 512]	128	0.5	10^{-3}	64	Adam
Deep	[2048, 1024, 512, 256]	128	0.4	5×10^{-4}	64	Adam
Regularized	[1024, 512]	128	0.5	10^{-3}	64	Adam
Optimized	[1536, 768, 384]	256	0.3	5×10^{-4}	128	Adam

Tabella 2: Dettagli di Training per Configurazioni DNN

Config	Epochs	Weight Decay	Label Smooth.	BatchNorm	Features
Baseline	10	0.0	0.0	✓	Standard config
No Dropout	10	0.0	0.0	✓	No dropout
No BatchNorm	10	0.0	0.0	✗	No batch norm
Deep	15	0.0	0.0	✓	4 hidden layers
Regularized	10	10^{-4}	0.1	✓	L2 + label smooth
Optimized	20	5×10^{-5}	0.05	✓	Best practices

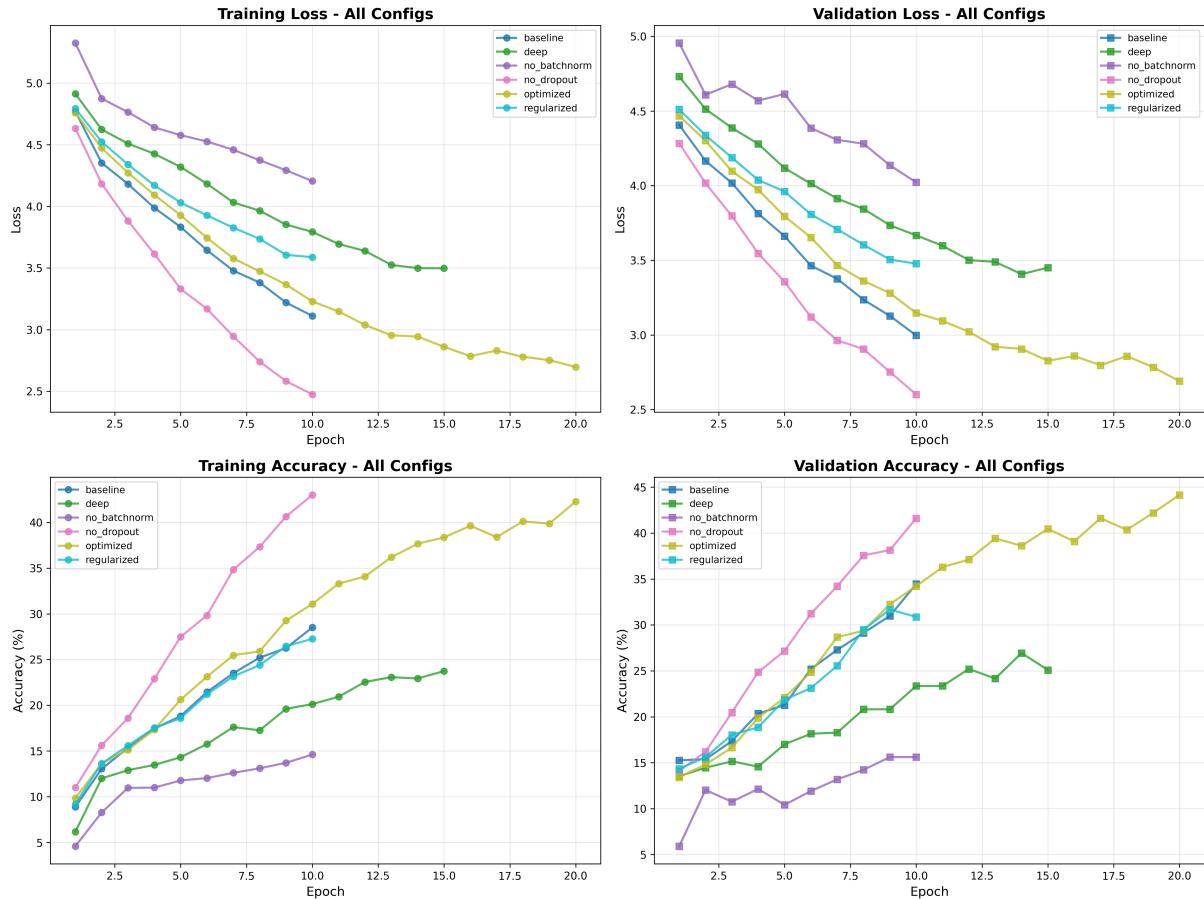


Figura 4: Evoluzione Loss e Accuracy delle DNN

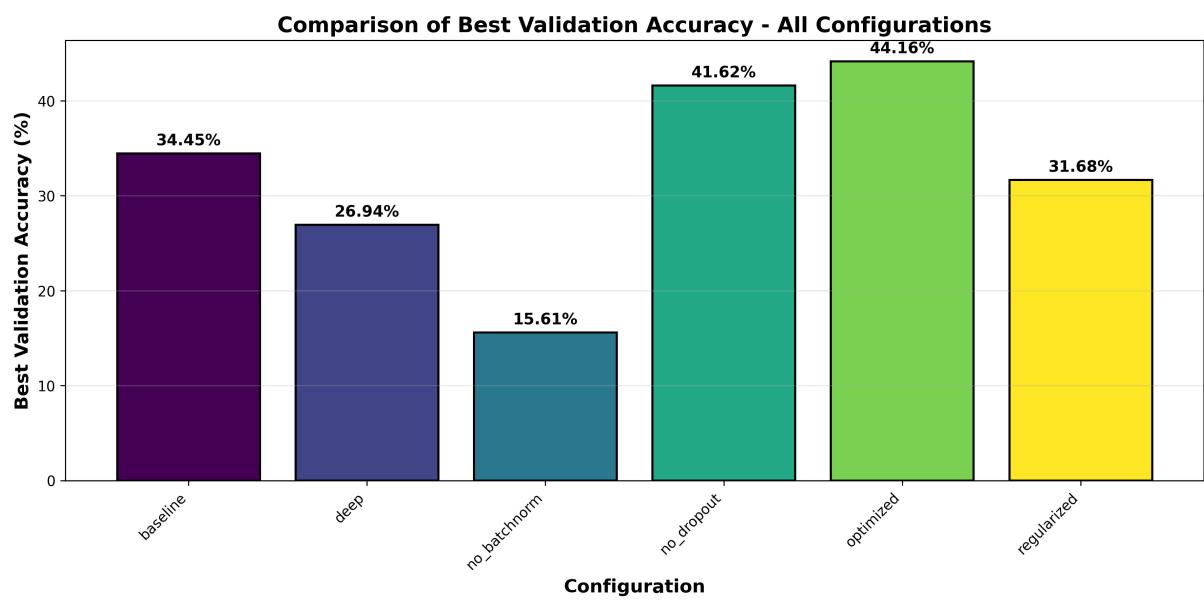


Figura 5: Confronto Accuracy delle DNN

7.3 Convolutional Neural Network (CNN)

7.3.1 Descrizione

È stata implementata una Convolutional Neural Network custom, progettata specificamente per il task di classificazione. Le CNN utilizzano layer convoluzionali che apprendono automaticamente feature gerarchiche dalle immagini, risultando particolarmente efficaci per task di computer vision.

È stata aggiunta la possibilità di utilizzare Residual Blocks per imitare la struttura di ResNet.

7.3.2 Motivazione

La CNN custom è stata sviluppata per:

- **Apprendimento pratico:** implementare da zero un'architettura convoluzionale
- **Confronto con DNN:** dimostrare la superiorità delle convoluzioni per dati spaziali
- **Ottimizzazione specifica:** adattare l'architettura alle caratteristiche specifiche del dataset

7.3.3 Implementazione

```

1  class FaceEmbeddingCNN(nn.Module):
2      """
3          Convolutional Neural Network per generare embeddings di volti.
4          Durante training: classificazione delle identità (cross entropy)
5          Durante inference: estrazione embedding dal penultimo layer
6
7          Supporta sia CNN standard che CNN con residual connections (ResNet-
8          style)
9          """
10
11     def __init__(self,
12                  input_channels=3,    # RGB
13                  num_filters=[32, 64, 128],   # filtri per ogni conv
14                  block
15                  kernel_sizes=[3, 3, 3],   # kernel size per ogni conv
16                  block
17                  fc_hidden_size=512,    # dimensione layer fully connected
18                  embedding_size=128,    # dimensione embedding finale
19                  num_classes=5749,    # numero identità
20                  dropout_rate=0.5,    # dropout rate
21                  use_batchnorm=True,   # usa batch normalization
22                  use_global_avg_pool=False, # usa global average
23                  pooling
24                  # usa residual connections (skip connections)
25                  use_residual=False):
26
27      """
28      Args:
29          input_channels: numero canali input (3 per RGB)
30          num_filters: lista con numero filtri per ogni conv block
            kernel_sizes: lista con kernel size per ogni conv block
            fc_hidden_size: dimensione layer fully connected dopo conv
            embedding_size: dimensione del vettore embedding
            num_classes: numero di identità (classi) per
            classificazione
            dropout_rate: probabilità dropout (0 = no dropout)

```

```

31         use_batchnorm: se True, aggiunge BatchNorm dopo ogni conv
32         use_global_avg_pool: se True, usa GAP invece di flatten
33         use_residual: se True, usa residual blocks con skip
34     connections
35     """
36
37     super(FaceEmbeddingCNN, self).__init__()
38
39     self.embedding_size = embedding_size
40     self.use_batchnorm = use_batchnorm
41     self.use_global_avg_pool = use_global_avg_pool
42     self.use_residual = use_residual
43
44     # Costruzione convolutional blocks
45     if use_residual:
46         # Residual CNN con skip connections
47         conv_layers = []
48         in_channels = input_channels
49
50         for num_filter, kernel_size in zip(num_filters, kernel_sizes):
51             # Residual block
52             conv_layers.append(ResidualBlock(
53                 in_channels,
54                 num_filter,
55                 kernel_size=kernel_size,
56                 dropout_rate=dropout_rate,
57                 use_batchnorm=use_batchnorm
58             ))
59
60             # MaxPool dopo ogni residual block
61             conv_layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
62
63             in_channels = num_filter
64
65     else:
66         # Standard CNN senza residual connections
67         conv_layers = []
68         in_channels = input_channels
69
70         for num_filter, kernel_size in zip(num_filters, kernel_sizes):
71             # Conv2d
72             conv_layers.append(nn.Conv2d(
73                 in_channels,
74                 num_filter,
75                 kernel_size=kernel_size,
76                 padding=kernel_size//2
77             ))
78
79             # BatchNorm2d
80             if use_batchnorm:
81                 conv_layers.append(nn.BatchNorm2d(num_filter))
82
83             # Activation
84             conv_layers.append(nn.ReLU())

```

```

85         # MaxPool2d
86         conv_layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
87     )
88
89     # Dropout2d
90     if dropout_rate > 0:
91         conv_layers.append(nn.Dropout2d(dropout_rate))
92
93     in_channels = num_filter
94
95     self.conv_blocks = nn.Sequential(*conv_layers)
96
97     # Calcola dimensione output dopo conv blocks
98     num_conv_blocks = len(num_filters)
99     spatial_size = 128 // (2 ** num_conv_blocks)
100
101    if use_global_avg_pool:
102        self.global_pool = nn.AdaptiveAvgPool2d(1)
103        flatten_size = num_filters[-1]
104    else:
105        self.global_pool = None
106        flatten_size = num_filters[-1] * spatial_size * spatial_size
107
108    # Fully connected layers per embedding
109    fc_layers = []
110
111    # FC hidden layer
112    fc_layers.append(nn.Linear(flatten_size, fc_hidden_size))
113    if use_batchnorm:
114        fc_layers.append(nn.BatchNorm1d(fc_hidden_size))
115    fc_layers.append(nn.ReLU())
116    fc_layers.append(nn.Dropout(dropout_rate))
117
118    # Embedding layer (penultimo layer)
119    fc_layers.append(nn.Linear(fc_hidden_size, embedding_size))
120    if use_batchnorm:
121        fc_layers.append(nn.BatchNorm1d(embedding_size))
122    fc_layers.append(nn.ReLU())
123
124    self.fc_blocks = nn.Sequential(*fc_layers)
125
126    # Classification head (solo per training)
127    self.classifier = nn.Linear(embedding_size, num_classes)

```

Listing 10: Classe CNN

```

1
2     def forward(self, x):
3         """
4             Forward pass completo per training (con classificazione)
5
6             Args:
7                 x: tensor [batch_size, channels, height, width]
8
9             Returns:
10                logits: tensor [batch_size, num_classes]
11            """
12            # Conv blocks

```

```

13     x = self.conv_blocks(x)
14
15     # Flatten o Global Average Pooling
16     if self.use_global_avg_pool:
17         x = self.global_pool(x)
18         x = x.view(x.size(0), -1)
19     else:
20         x = x.view(x.size(0), -1)
21
22     # FC blocks per embedding
23     embedding = self.fc_blocks(x)
24
25     # Classification
26     logits = self.classifier(embedding)
27
28     return logits

```

Listing 11: Forward della classe CNN

```

1 class ResidualBlock(nn.Module):
2     """
3         Blocco residuale per CNN
4         Conv -> BN -> ReLU -> Conv -> BN -> (+skip) -> ReLU
5     """
6
7     def __init__(self, in_channels, out_channels, kernel_size=3,
8                  dropout_rate=0.5, use_batchnorm=True):
9         super(ResidualBlock, self).__init__()
10
11         padding = kernel_size // 2
12
13         # Prima convoluzione
14         self.conv1 = nn.Conv2d(in_channels, out_channels,
15                               kernel_size=kernel_size, padding=padding)
16         self.bn1 = nn.BatchNorm2d(
17             out_channels) if use_batchnorm else nn.Identity()
18         self.relu1 = nn.ReLU()
19
20         # Seconda convoluzione
21         self.conv2 = nn.Conv2d(out_channels, out_channels,
22                               kernel_size=kernel_size, padding=padding)
23         self.bn2 = nn.BatchNorm2d(
24             out_channels) if use_batchnorm else nn.Identity()
25
26         # Projection shortcut se dimensioni diverse
27         if in_channels != out_channels:
28             self.shortcut = nn.Sequential(
29                 nn.Conv2d(in_channels, out_channels, kernel_size=1),
30                 nn.BatchNorm2d(
31                     out_channels) if use_batchnorm else nn.Identity()
32             )
33         else:
34             self.shortcut = nn.Identity()
35
36         self.relu2 = nn.ReLU()
37         self.dropout = nn.Dropout2d(
38             dropout_rate) if dropout_rate > 0 else nn.Identity()
39
40     def forward(self, x):

```

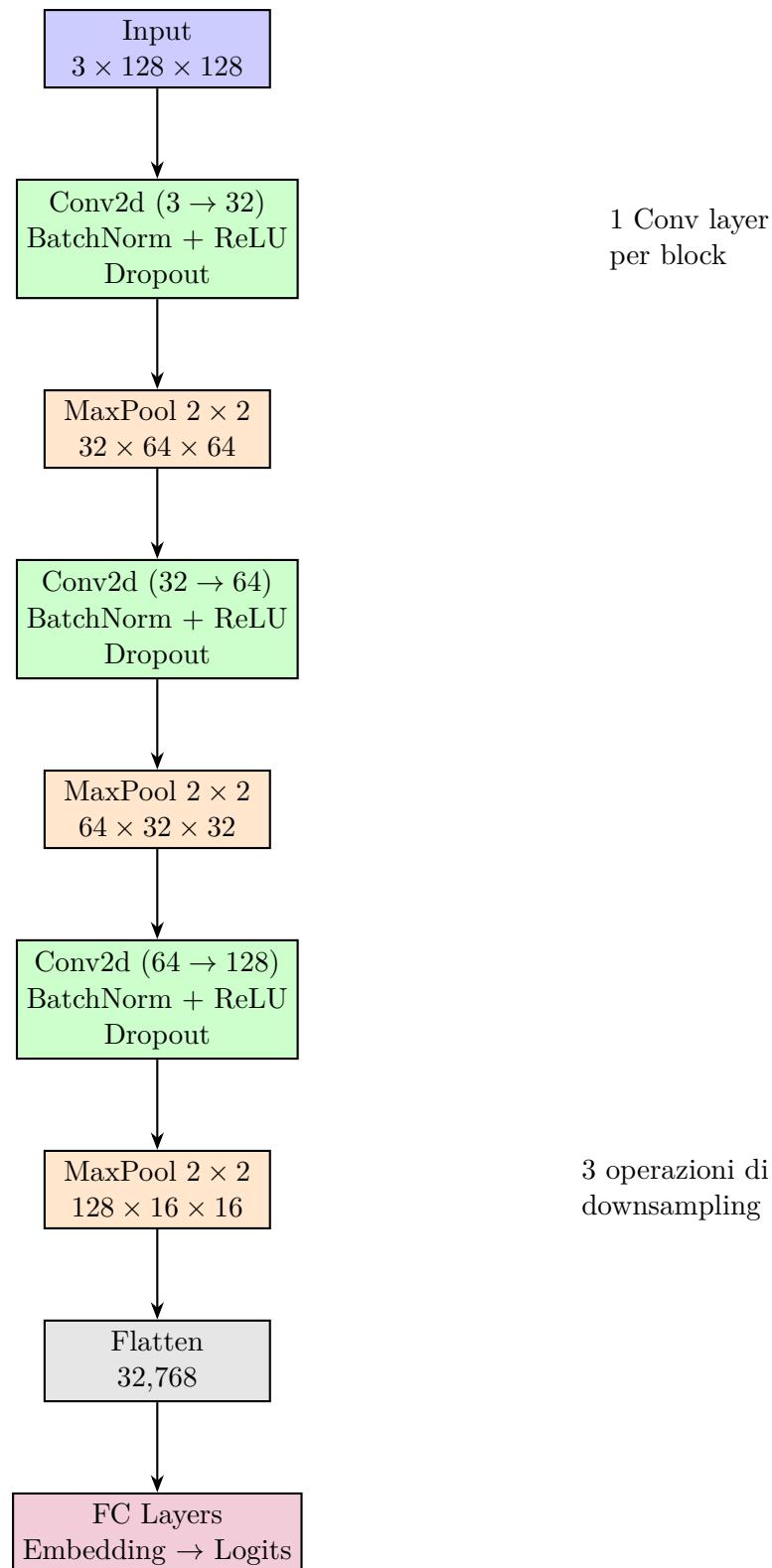
```
41     identity = self.shortcut(x)
42
43     out = self.conv1(x)
44     out = self.bn1(out)
45     out = self.relu1(out)
46
47     out = self.conv2(out)
48     out = self.bn2(out)
49
50     # Skip connection
51     out = out + identity
52     out = self.relu2(out)
53     out = self.dropout(out)
54
55     return out
```

Listing 12: Classe ResidualBlock per creare blocchi di layer convoluzionali con skip connection

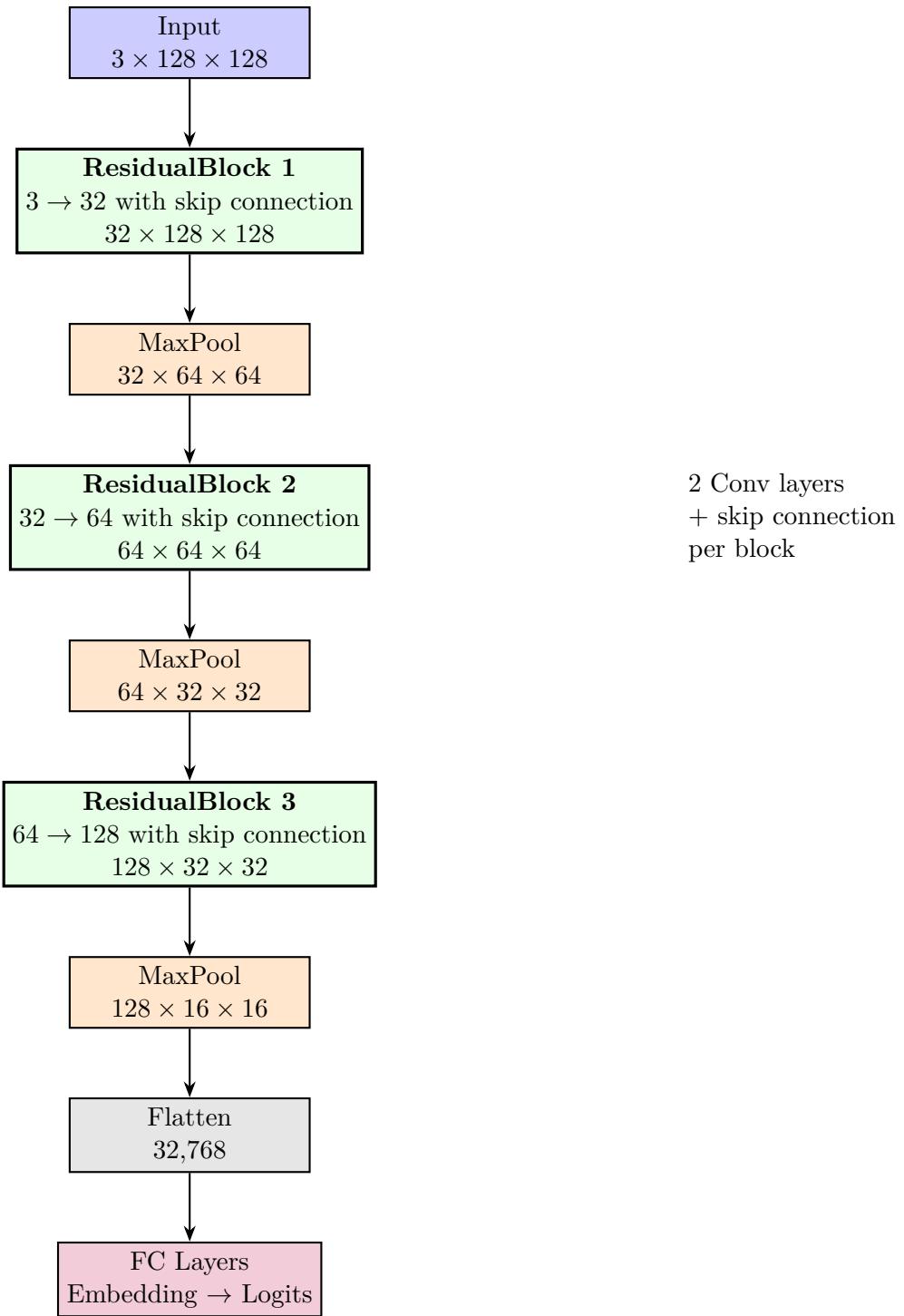
7.3.4 Dettagli Tecnici

- **Architettura:** Conv2D → BatchNorm → ReLU → MaxPool → ... → FC
- Possibilità di aggiungere Residual Blocks, imitando la struttura di ResNet
- **Dataset utilizzati:** sono stati utilizzati due dataset: LFW e CelebA; su entrambi sono state testate tutte le configurazioni.

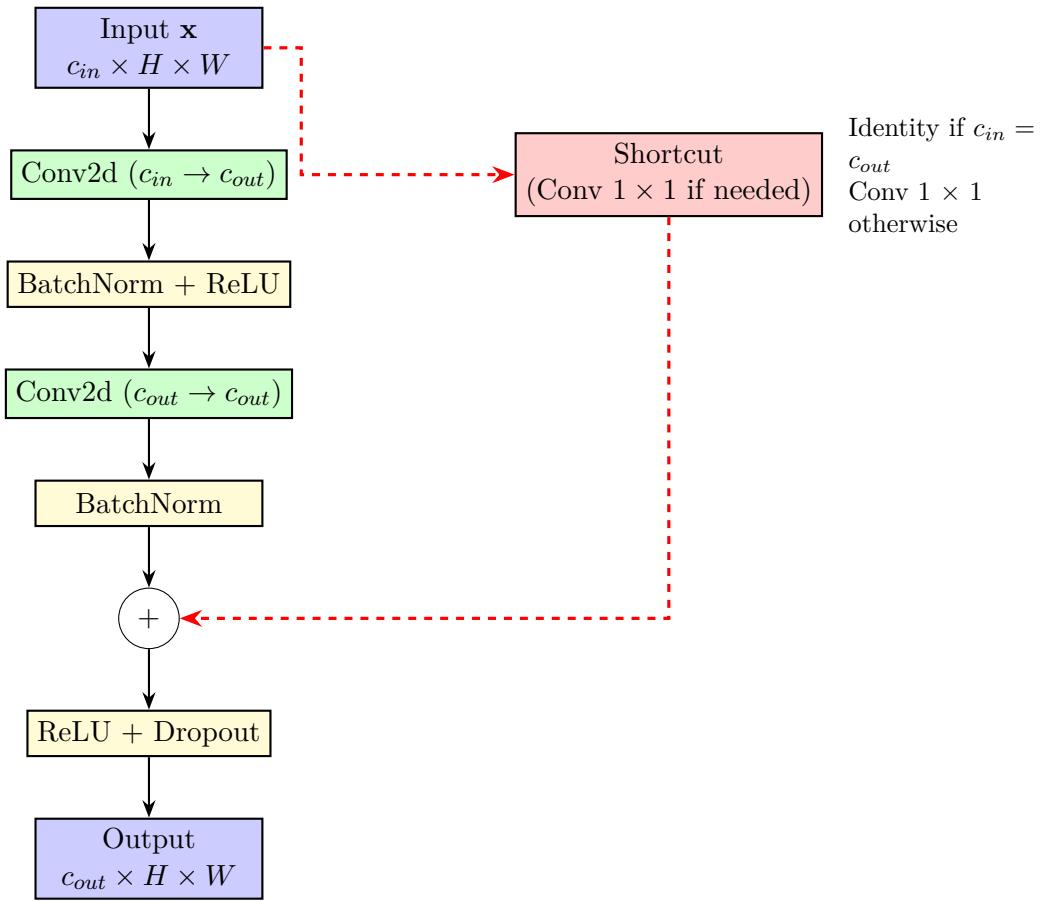
7.3.5 Architettura CNN classica (use_residual=false)



7.3.6 CNN con Residual Blocks (use_residual=true)



7.3.7 ResidualBlock Struttura Interna



7.3.8 Configurazioni Testate

Sono state esplorate diverse configurazioni architetturali:

Tabella 3: Configurazioni CNN per Face Recognition

Config	Conv Layers	Filters	Kernels	FC Hidden	Embedding	Dropout	LR	Batch Size
Baseline	3	[32, 64, 128]	[3, 3, 3]	512	128	0.5	10^{-3}	64
Shallow	2	[32, 64]	[3, 3]	512	128	0.4	10^{-3}	64
Deep	4	[32, 64, 128, 256]	[3, 3, 3, 3]	1024	256	0.4	5×10^{-4}	32
Large Kernels	3	[32, 64, 128]	[5, 5, 5]	512	128	0.5	10^{-3}	64
GAP	3	[64, 128, 256]	[3, 3, 3]	512	128	0.4	10^{-3}	64
Optimized	4	[64, 128, 256, 512]	[3, 3, 3, 3]	1024	256	0.3	5×10^{-4}	64
Deep Residual	6	[64, 64, 128, 128, 256, 256]	[3, 3, 3, 3, 3, 3]	1024	256	0.3	3×10^{-4}	32

Tabella 4: Dettagli di Training per Configurazioni CNN

Config	Epochs	Weight Decay	Label Smooth.	BatchNorm	Features
Baseline	45	0.0	0.0	✓	—
Shallow	45	0.0	0.0	✓	Fewer layers
Deep	45	10^{-5}	0.05	✓	More layers
Large Kernels	45	0.0	0.0	✓	5×5 kernels
GAP	35	10^{-5}	0.0	✓	Global Avg Pool
Optimized	35	10^{-4}	0.1	✓	Best practices
Deep Residual	35	10^{-4}	0.1	✓	Skip connections

Tabella 5: Convolutional Neural Network performance su LFW dataset

Configurazione	Parametri	Migliore Val Acc (%)	Epoche*
cnn_baseline	17,679,989	57.80	45
cnn_deep	18,910,069	71.56	45
cnn_deep_residual	5,532,725	52.25	35
cnn_gap	1,311,861	13.06	9
cnn_large_kernels	17,845,365	60.81	45
cnn_optimized	36,850,805	78.84	35
cnn_shallow	34,383,093	64.16	45

*Valore < 45 indica early stopping

Tabella 6: Convolutional Neural Network performance su CelebA dataset

Configurazione	Parametri	Migliore Val Acc (%)	Epoche*
cnn_baseline	17,679,989	80.88	30
cnn_deep	18,910,069	81.73	30
cnn_deep_residual	5,532,725	82.65	30
cnn_gap	1,311,861	79.03	30
cnn_optimized	36,850,805	82.40	30
cnn_shallow	34,383,093	81.34	30

*Valore < 30 indica early stopping

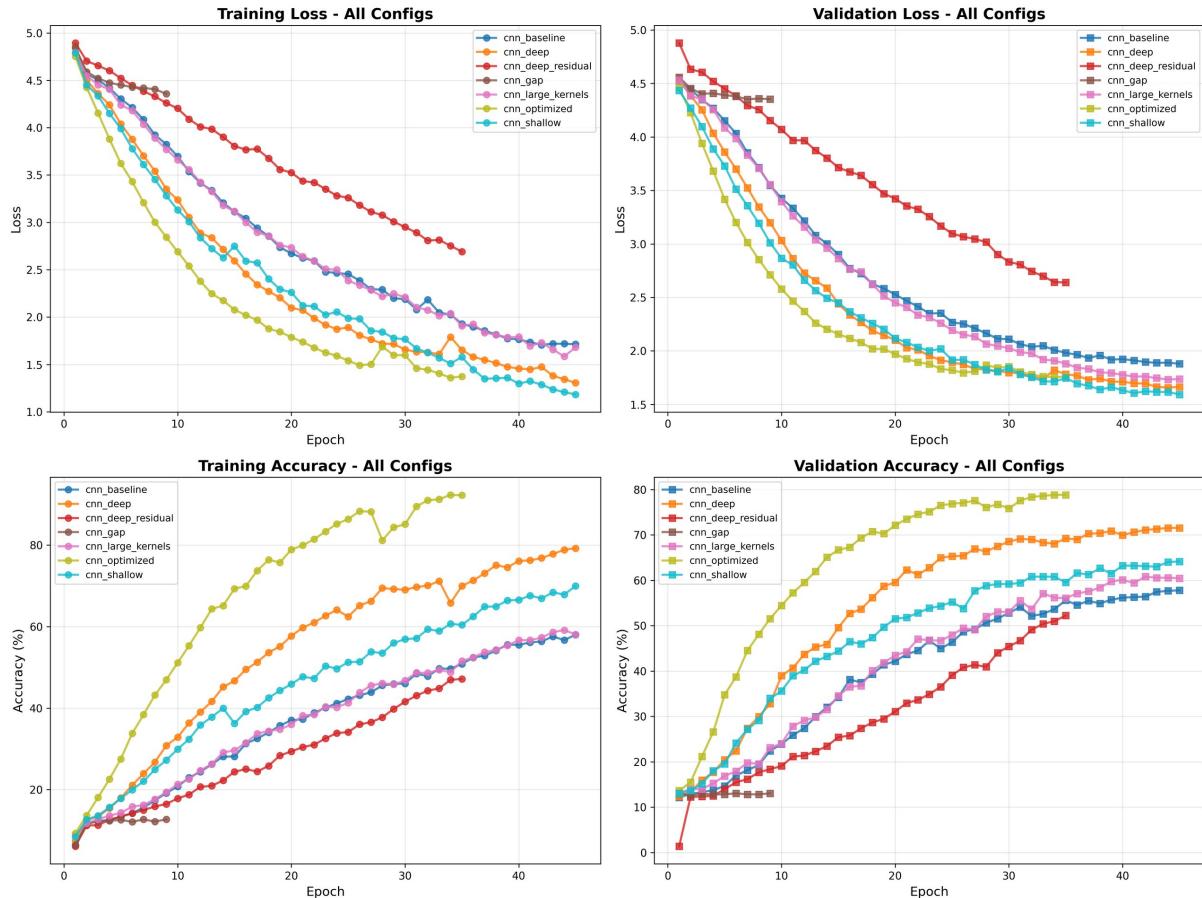


Figura 6: Evoluzione Loss e Accuracy delle CNN (LFW)

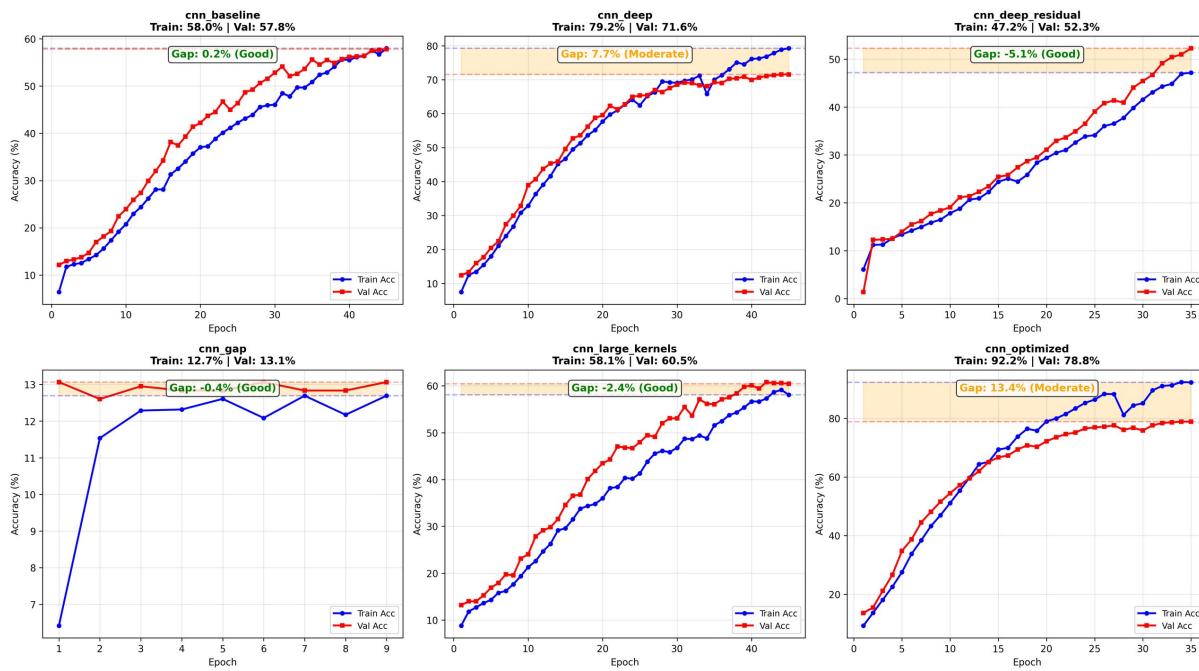


Figura 7: Analisi overfitting delle CNN (LFW)

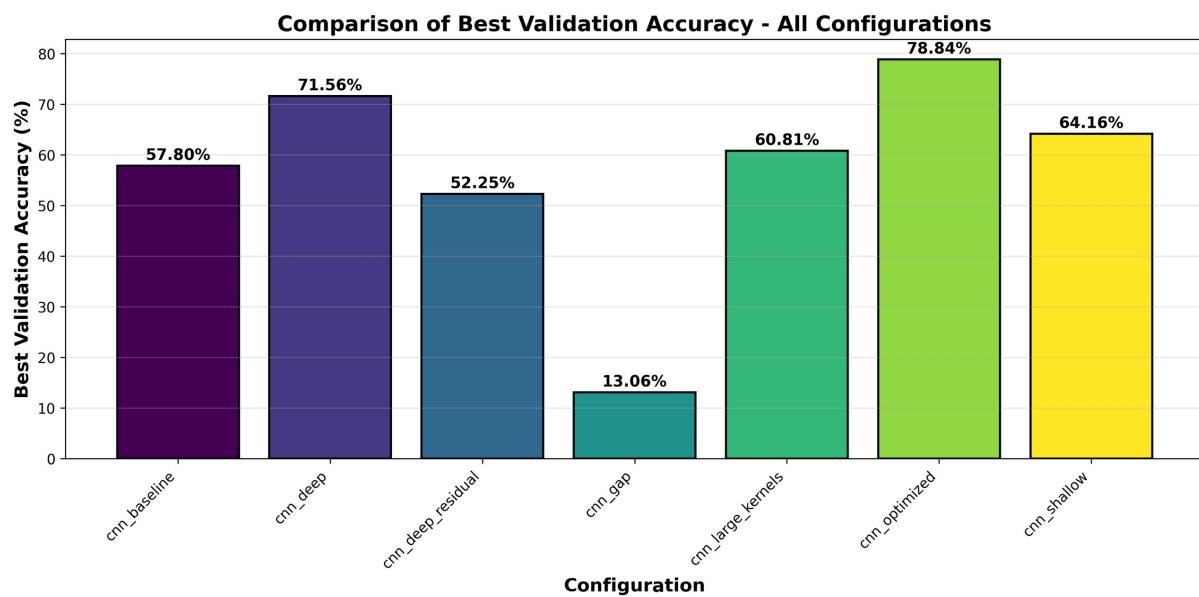


Figura 8: Confronto Accuracy delle CNN (LFW)

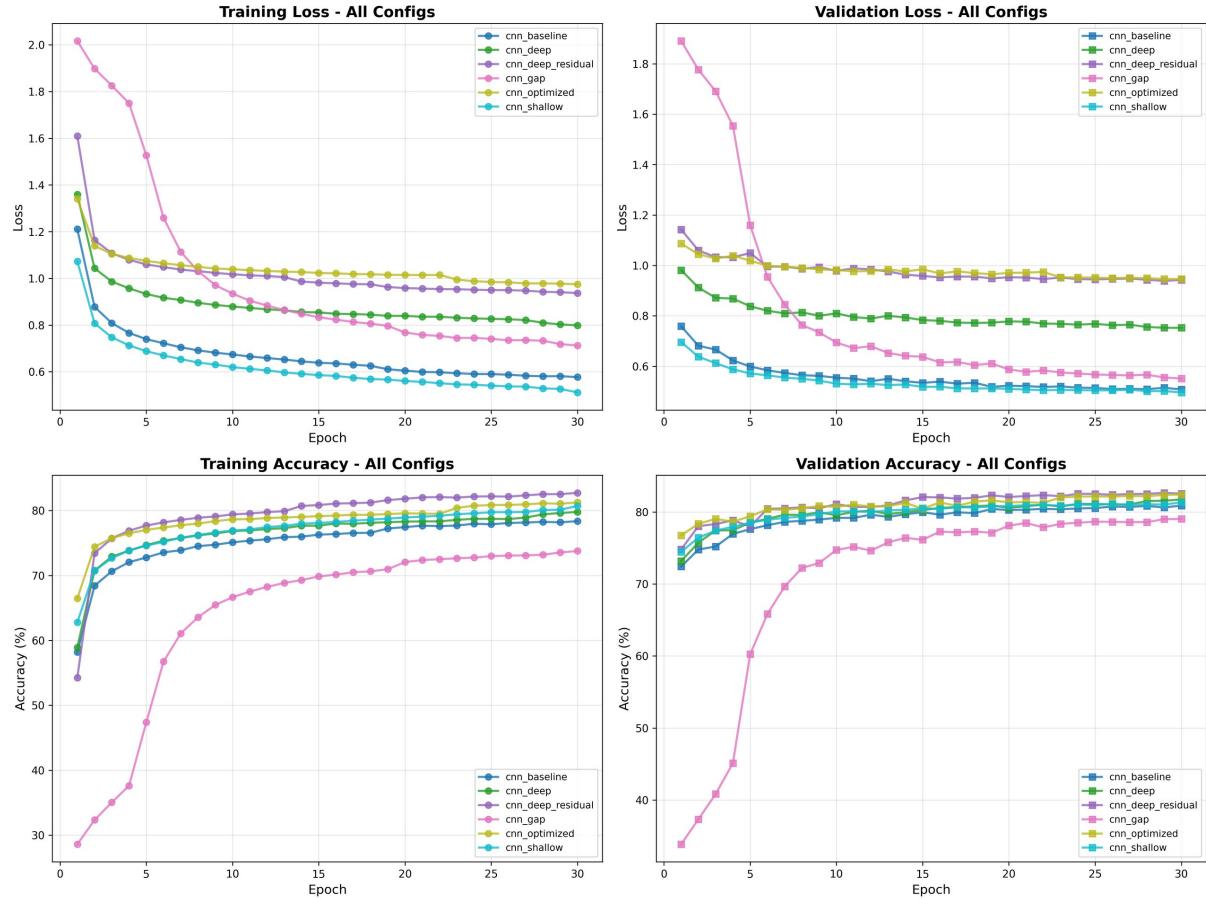


Figura 9: Evoluzione Loss e Accuracy delle CNN (CelebA)

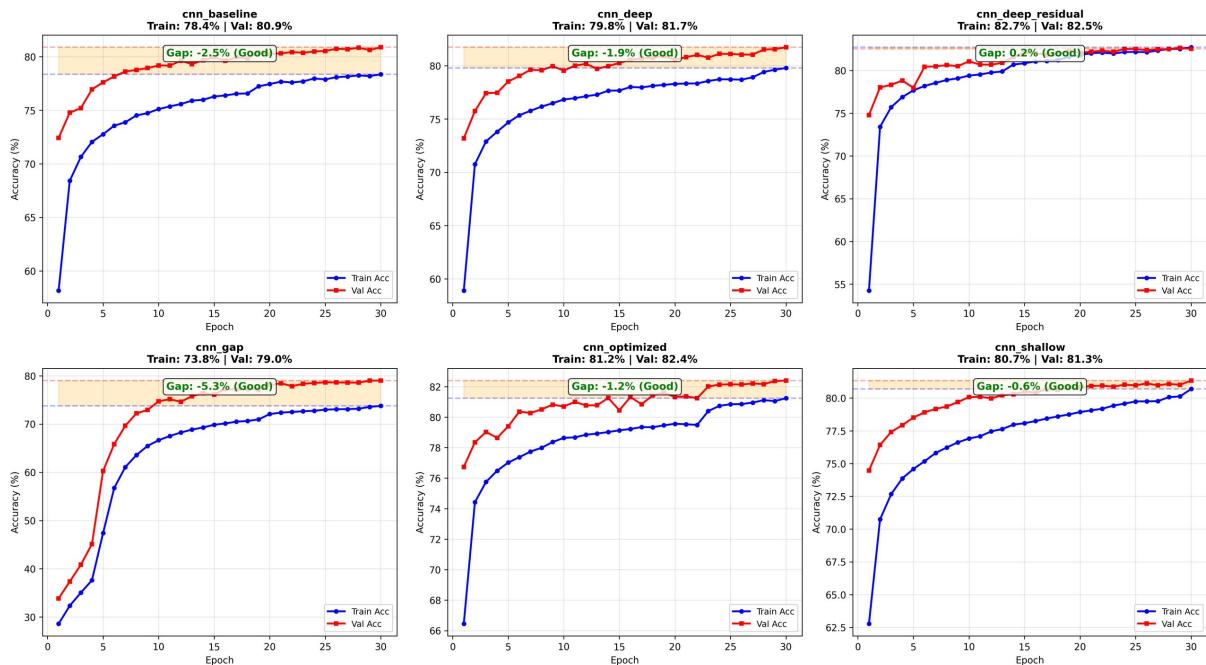


Figura 10: Analisi overfitting delle CNN (CelebA)

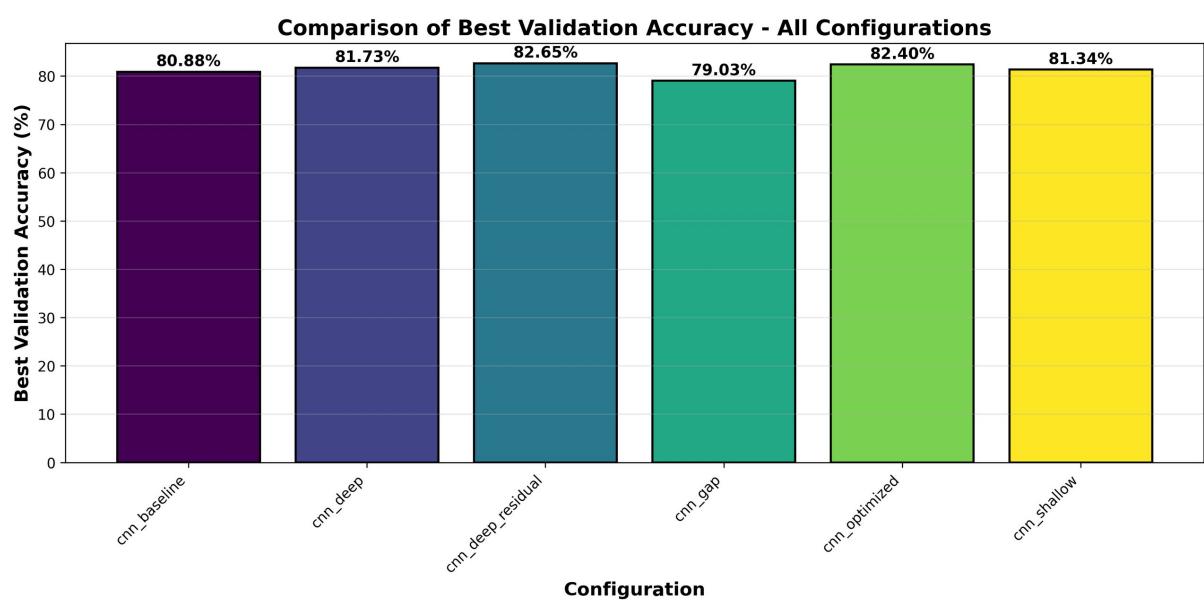


Figura 11: Confronto Accuracy delle CNN (CelebA)

7.4 Transfer Learning e Fine-Tuning

7.4.1 Descrizione

Sono state usate tecniche di Transfer Learning partendo da un modello pre-addestrato su ImageNet, seguito da una fase di fine-tuning sui dati specifici dell'applicazione. Questo approccio sfrutta la conoscenza appresa su dataset di grandi dimensioni per migliorare le prestazioni su task specifici con dataset limitati.

7.4.2 Motivazione

Il transfer learning e fine-tuning sono stati implementati per:

- **Migliori prestazioni:** ottenere risultati superiori rispetto ai modelli trainati da zero
- **Efficienza:** ridurre drasticamente il tempo di training necessario
- **Dataset limitato:** superare il problema di scarsità di dati di training
- **Best practice industriale:** utilizzare l'approccio standard nell'industria
- **Confronto con CNN custom:** valutare il vantaggio del pre-training

7.4.3 Implementazione

```

1 class TransferLearningEmbeddingCNN(nn.Module):
2     """
3         Convolutional Neural Network per generare embeddings di volti.
4         Durante training: classificazione delle identità (cross entropy)
5         Durante inference: estrazione embedding dal penultimo layer
6
7         Supporta sia CNN standard che CNN con residual connections (ResNet-
8         style)
9         """
10
11    def __init__(self,
12                  input_channels=3, # RGB
13                  num_filters=[32, 64, 128], # filtri per ogni conv
14                  block
15                  kernel_sizes=[3, 3, 3], # kernel size per ogni conv
16                  block
17                  fc_hidden_size=512, # dimensione layer fully connected
18                  embedding_size=128, # dimensione embedding finale
19                  num_classes=5749, # numero identità
20                  dropout_rate=0.5, # dropout rate
21                  use_batchnorm=True, # usa batch normalization
22                  use_global_avg_pool=False, # usa global average
23                  pooling
24                  # usa residual connections (skip connections)
25                  use_residual=False,
26                  use_pretrained_resnet=False, # <se usare il
27                  transferlearning con resnet
28                  resnet_version='resnet18', # versione resnet usata
29                  freeze_resnet_backbone=False): # per freezzare il
30          backbone o no
31          """
32          Args:
33              input_channels: numero canali input (3 per RGB)
34              num_filters: lista con numero filtri per ogni conv block

```

```

29         kernel_sizes: lista con kernel size per ogni conv block
30         fc_hidden_size: dimensione layer fully connected dopo conv
31         embedding_size: dimensione del vettore embedding
32         num_classes: numero di identita' (classi) per
33             classificazione
34             dropout_rate: probabilita' dropout (0 = no dropout)
35             use_batchnorm: se True, aggiunge BatchNorm dopo ogni conv
36             use_global_avg_pool: se True, usa GAP invece di flatten
37             use_residual: se True, usa residual blocks con skip
38             connections
39             """
40
41     super(TransferLearningEmbeddingCNN, self).__init__()
42
43     self.embedding_size = embedding_size
44     self.use_batchnorm = use_batchnorm
45     self.use_global_avg_pool = use_global_avg_pool
46     self.use_residual = use_residual
47
48     # Costruzione convolutional blocks
49     if use_pretrained_resnet:
50         # TRANSFER LEARNING: ResNet pre-trained
51         if resnet_version == 'resnet18':
52             backbone = models.resnet18(pretrained=True)
53             backbone_output_channels = 512
54         elif resnet_version == 'resnet34':
55             backbone = models.resnet34(pretrained=True)
56             backbone_output_channels = 512
57         elif resnet_version == 'resnet50':
58             backbone = models.resnet50(pretrained=True)
59             backbone_output_channels = 2048
60         else:
61             raise ValueError(
62                 "resnet_version deve essere 'resnet18', 'resnet34' o
63                 'resnet50'")
64
65         # Togli avgpool + fc finale
66         self.backbone = nn.Sequential(*list(backbone.children()))
67
68         # Congela se richiesto
69         if freeze_resnet_backbone:
70             for param in self.backbone.parameters():
71                 param.requires_grad = False
72
73         # Aggiungo i miei residual blocks dopo ResNet
74         residual_layers = []
75         in_channels = backbone_output_channels
76
77         for num_filter, kernel_size in zip(num_filters, kernel_sizes):
78             residual_layers.append(ResidualBlock(
79                 in_channels,
80                 num_filter,
81                 kernel_size=kernel_size,
82                 dropout_rate=dropout_rate,
83                 use_batchnorm=use_batchnorm
84             ))
85             in_channels = num_filter

```

```

82         self.conv_blocks = nn.Sequential(*residual_layers)
83
84
85     # Calcola dimensione output dopo conv blocks
86     if use_pretrained_resnet:
87         # Con ResNet usa sempre global avg pool
88         self.use_global_avg_pool = True
89         self.global_pool = nn.AdaptiveAvgPool2d(1)
90         flatten_size = num_filters[-1]
91
92
93     # Fully connected layers per embedding
94     fc_layers = []
95
96     # FC hidden layer
97     fc_layers.append(nn.Linear(flatten_size, fc_hidden_size))
98     if use_batchnorm:
99         fc_layers.append(nn.BatchNorm1d(fc_hidden_size))
100    fc_layers.append(nn.ReLU())
101    fc_layers.append(nn.Dropout(dropout_rate))
102
103    # Embedding layer (penultimo layer)
104    fc_layers.append(nn.Linear(fc_hidden_size, embedding_size))
105    if use_batchnorm:
106        fc_layers.append(nn.BatchNorm1d(embedding_size))
107        fc_layers.append(nn.ReLU())
108
109    self.fc_blocks = nn.Sequential(*fc_layers)
110
111    # Classification head (solo per training)
112    self.classifier = nn.Linear(embedding_size, num_classes)

```

Listing 13: Classe Transfer Learning Model

7.4.4 Dettagli Tecnici

- **Base model:** ResNet18, ResNet34, ResNet50
- **Dataset di pre-training:** ImageNet (1.4M immagini, 1000 classi)
- **Strategia di fine-tuning:**
 - Feature extraction: freeze tutti i layer base
 - Fine-tuning parziale: unfreeze ultimi N layer
 - Fine-tuning completo: unfreeze tutti i layer

7.4.5 Configurazioni Testate

Sono state confrontate diverse strategie di transfer learning e fine-tuning:

Tabella 7: Configurazioni Transfer Learning (ResNet) per Face Recognition

Config	ResNet Version	Frozen Backbone	Custom Filters	FC Hidden	Embedding Size	Dropout	LR	Batch Size
ResNet18 Frozen	ResNet18	✓	[256, 256]	512	128	0.3	10^{-3}	64
ResNet18 Finetuned	ResNet18	✗	[256, 256]	512	128	0.3	10^{-4}	32
ResNet34 Frozen	ResNet34	✓	[384, 256]	768	256	0.3	10^{-3}	48
ResNet50 Frozen	ResNet50	✓	[512, 256]	1024	256	0.4	10^{-3}	32

Tabella 8: Dettagli di Training per Configurazioni Transfer Learning

Config	Epoche	Weight Decay	Label Smooth.	GAP	Features
ResNet18 Frozen	35	10^{-4}	0.1	✓	Frozen backbone
ResNet18 Finetuned	35	10^{-4}	0.1	✓	Full fine-tuning
ResNet34 Frozen	35	10^{-4}	0.1	✓	Larger model
ResNet50 Frozen	35	10^{-4}	0.1	✓	Max capacity

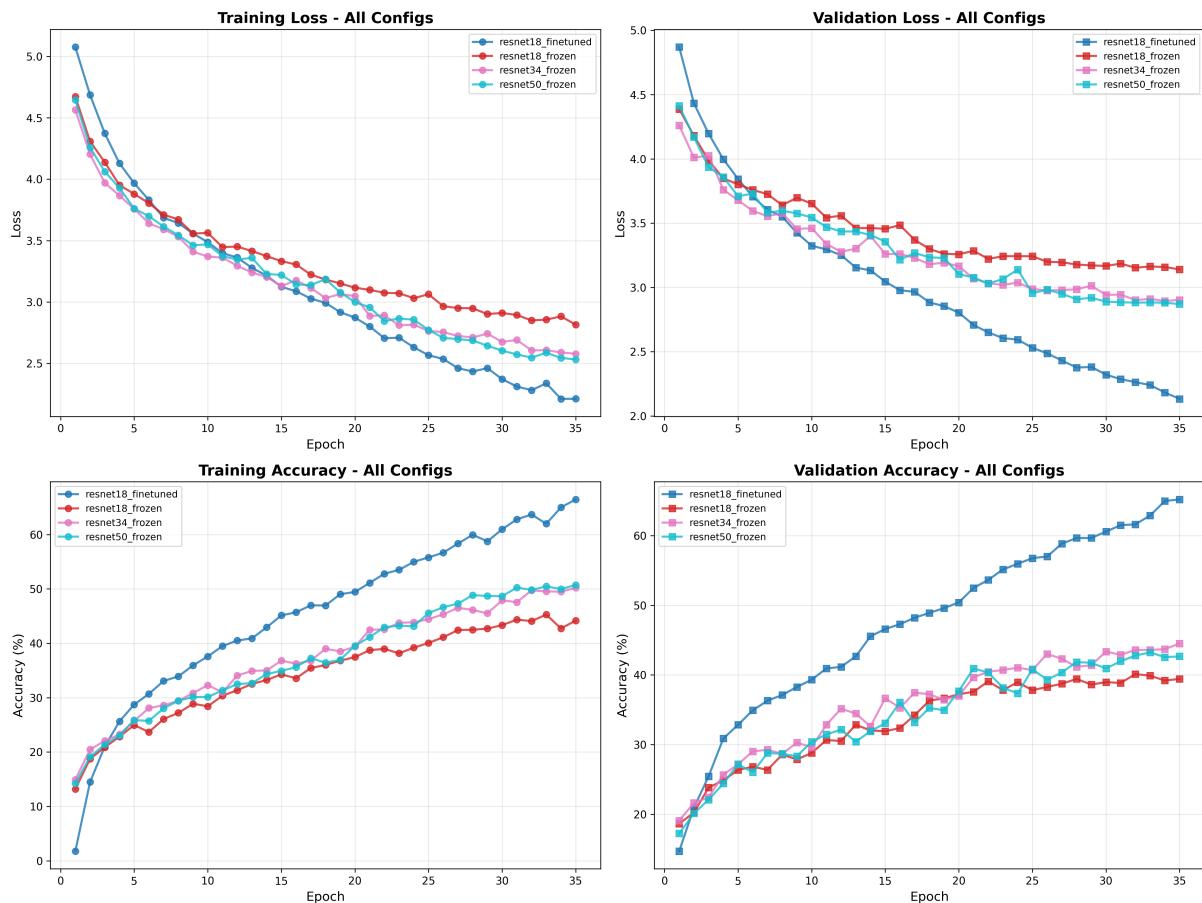


Figura 12: Evoluzione Loss e Accuracy delle CNN con Transfer Learning e Fine Tuning (LFW)

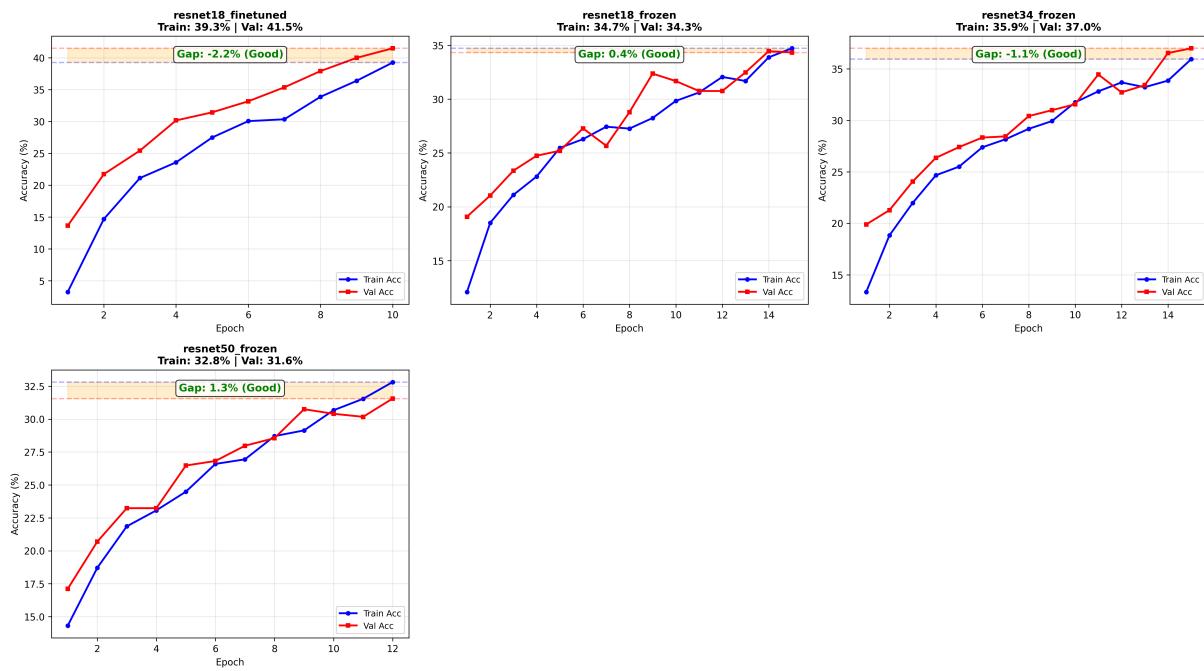


Figura 13: Analisi overfitting delle CNN con Transfer Learning e Fine Tuning (LFW)

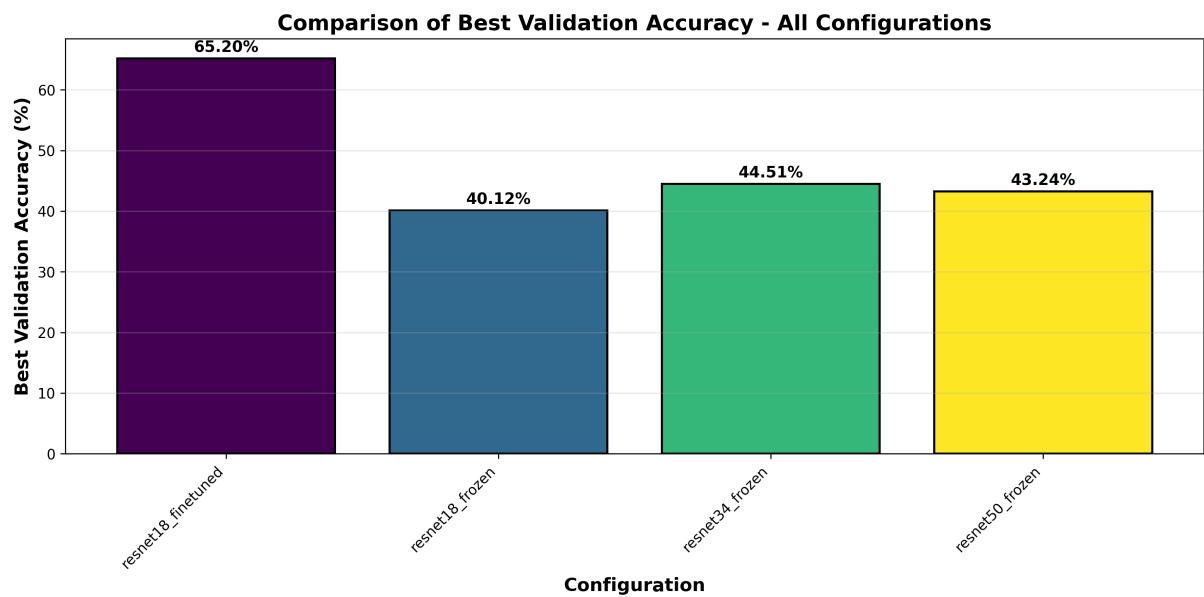


Figura 14: Confronto Accuracy delle CNN con Transfer Learning e Fine Tuning (LFW)

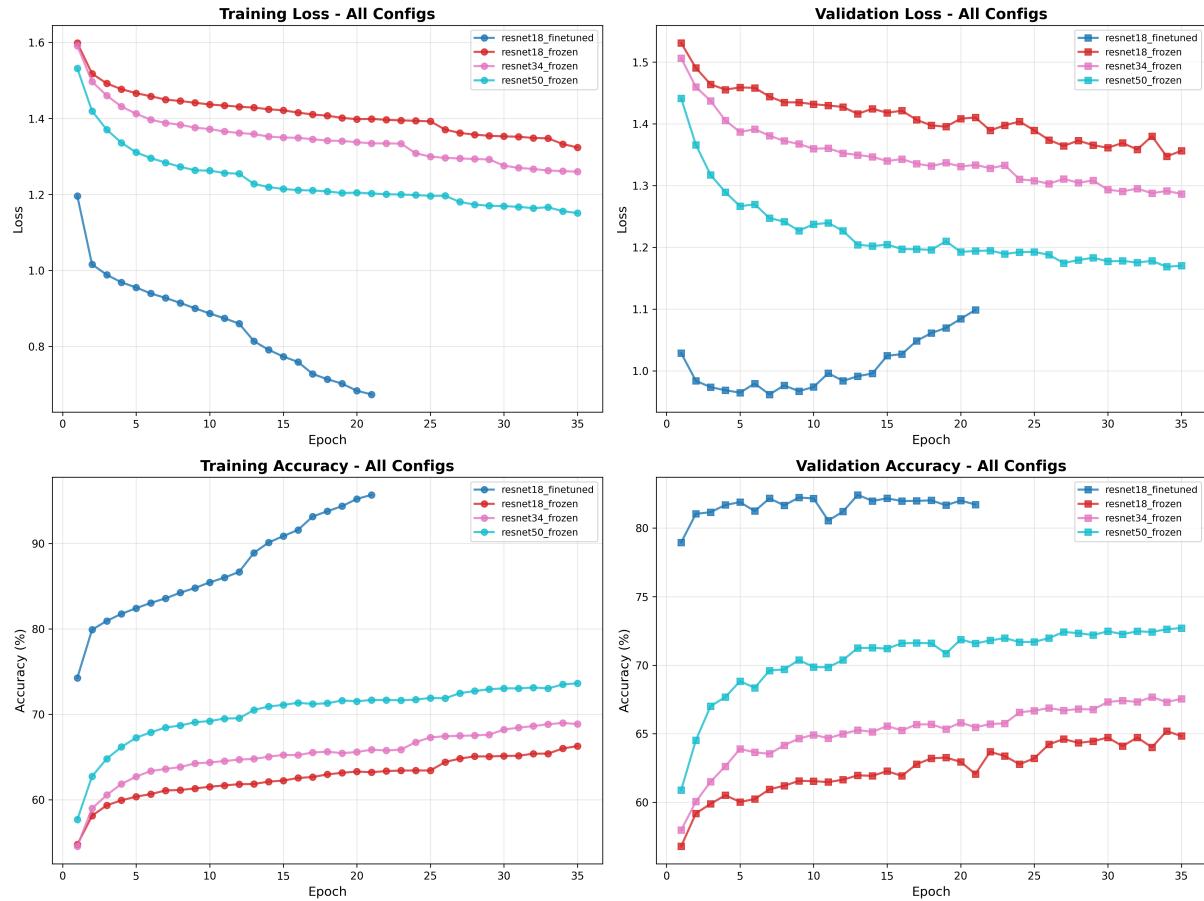


Figura 15: Evoluzione Loss e Accuracy delle CNN con Transfer Learning e Fine Tuning (CelebA)

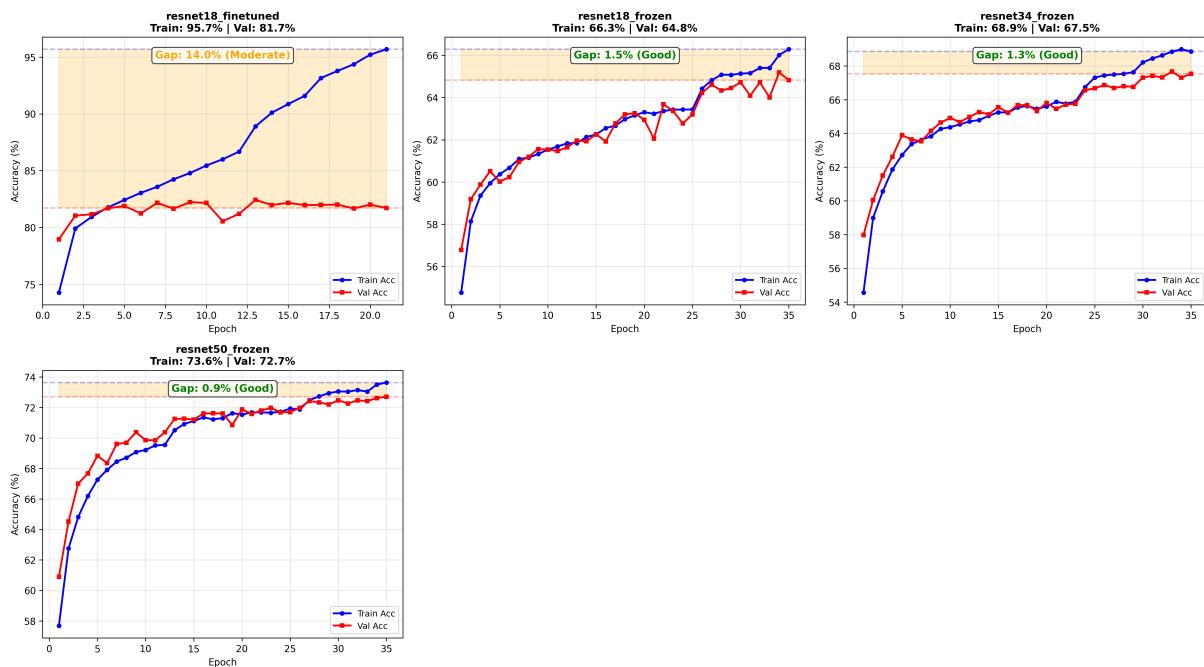


Figura 16: Analisi overfitting delle CNN con Transfer Learning e Fine Tuning (CelebA)

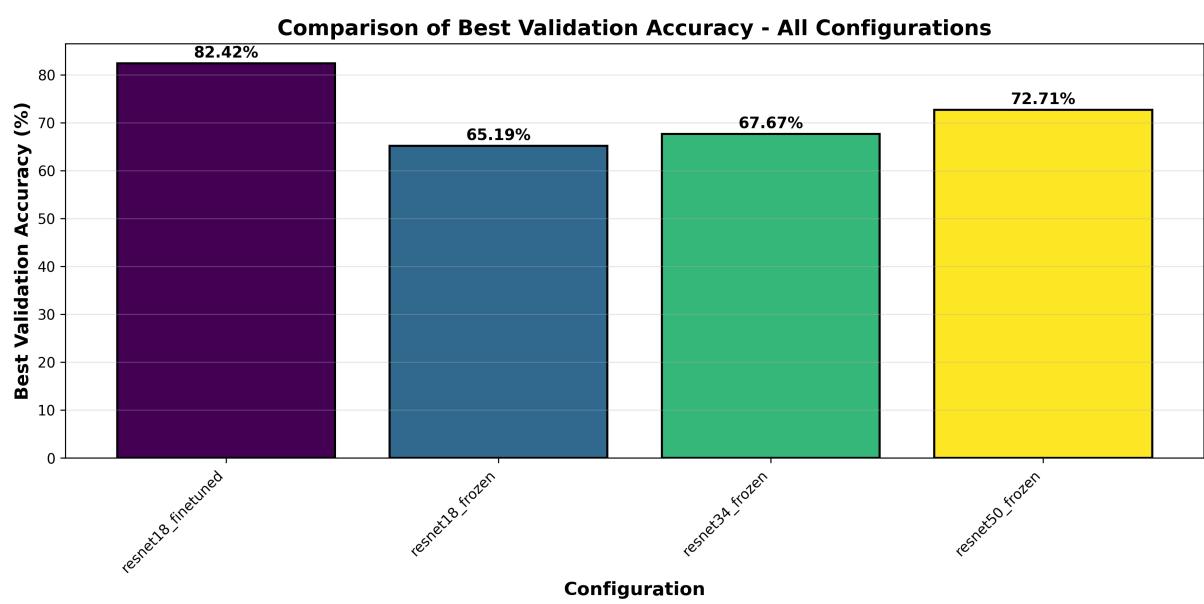


Figura 17: Confronto Accuracy delle CNN con Transfer Learning e Fine Tuning (CelebA)

7.5 Training delle architetture

- Per allenare le reti neurali è stato utilizzato il seguente codice (Nota: nei Colab sono presenti diversi metodi, simili uno all'altro, questo perchè il metodo è evoluto nel tempo insieme alle architetture, continuando a mantenere retrocompatibilità)
 - Nei Colab sono presenti due funzioni che utilizzano la funzione di training, una per una singola istanza (utilizzata per test principalmente) e quella maggiormente utilizzata che legge tutte le configurazioni e implementa early stopping. Riporto il codice di quest'ultima.

```
1 def train_multiple_configs(model_class, train_loader, test_loader,
2     num_classes,
3     checkpoint_dir='./checkpoints', device=None):
4     """
5     Train modelli con tutte le configurazioni predefinite
6
7     Args:
8         model_class: classe del modello
9         train_loader: DataLoader training
10        test_loader: DataLoader validation
11        num_classes: numero di classi
12        checkpoint_dir: directory per checkpoints
13        device: device (cuda/cpu)
14
15    Returns:
16        results: dict con risultati di ogni config
17    """
18
19    configs = get_cnn_configs()
20    results = {}
21
22    for config_name, config in configs.items():
23        print(f"Starting training: {config_name}")
24
25        try:
26            # Train con configurazione base
27            model, history, best_val_acc = train_model_with_config(
28                model_class=model_class,
29                config_name=config_name,
30                config=config,
31                train_loader=train_loader,
32                test_loader=test_loader,
33                num_classes=num_classes,
34                checkpoint_dir=checkpoint_dir,
35                device=device
36            )
37
38            # Fine-tuning per configurazioni ResNet frozen
39            if config_name.endswith('_frozen') and 'resnet' in config_name:
40                print(f"Starting FINE-TUNING phase: {config_name}")
41
42                # Scongela e fine-tune con LR molto basso
43                model, history_ft = unfreeze_and_finetune(
44                    model, train_loader, test_loader,
45                    num_epochs=5, # Poche epoche per fine-tuning
46                    learning_rate=1e-5, # LR molto basso
47                    checkpoint_dir=os.path.join(checkpoint_dir,
48                        config_name),
49                    device=device)
```

```

46             device=device
47         )
48
49         # Aggiorna best accuracy se migliorata
50         best_val_acc_ft = max(history_ft['val_acc'])
51         if best_val_acc_ft > best_val_acc:
52             best_val_acc = best_val_acc_ft
53             print(f"Fine-tuning migliorato: {best_val_acc:.2f}%"
54         )
55
56         results[config_name] = {
57             'best_val_acc': best_val_acc,
58             'history': history,
59             'config': config
60         }
61
62     except Exception as e:
63         print(f"\nERROR training {config_name}: {e}")
64         results[config_name] = {'error': str(e)}
65
66     return results

```

Listing 14: Funzione per fare training dei modelli su tutte le configurazioni

```

1 def train_model_with_config(model_class, config_name, config,
2                             train_loader, test_loader,
3                             num_classes, checkpoint_dir='./checkpoints',
4                             device=None):
5     """
6     Train un modello con una specifica configurazione
7
8     Args:
9         model_class: classe del modello (es. create_dnn_model)
10        config_name: nome della configurazione
11        config: dict con model_config e training_config
12        train_loader: DataLoader training
13        test_loader: DataLoader validation
14        num_classes: numero di classi
15        checkpoint_dir: directory base per checkpoints
16        device: device (cuda/cpu)
17
18    Returns:
19        model, history, best_val_acc
20    """
21
22    # Setup device
23    if device is None:
24        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
25
26    # Estrai configurazioni
27    model_config = config['model_config'].copy()
28    model_config['num_classes'] = num_classes
29    training_config = config['training_config']
30
31    # Crea modello
32    model = model_class(model_config)
33    model = model.to(device)
34
35    # Loss function (con label smoothing opzionale)

```

```

33     criterion = nn.CrossEntropyLoss(label_smoothing=training_config['label_smoothing'])
34
35     # Optimizer
36     if training_config['optimizer'] == 'adam':
37         optimizer = optim.Adam(
38             model.parameters(),
39             lr=training_config['learning_rate'],
40             weight_decay=training_config['weight_decay']
41         )
42     elif training_config['optimizer'] == 'rmsprop':
43         optimizer = optim.RMSprop(
44             model.parameters(),
45             lr=training_config['learning_rate'],
46             weight_decay=training_config['weight_decay']
47         )
48     else: # sgd
49         optimizer = optim.SGD(
50             model.parameters(),
51             lr=training_config['learning_rate'],
52             momentum=0.9,
53             weight_decay=training_config['weight_decay']
54         )
55
56     # Learning rate scheduler
57     scheduler = optim.lr_scheduler.ReduceLROnPlateau(
58         optimizer, mode='max', factor=0.5, patience=2
59     )
60
61     # Early stopping
62     early_stopping = EarlyStopping(
63         patience = training_config['early_stopping_patience'],
64         verbose=True
65     )
66
67     # Directory per questa configurazione
68     config_checkpoint_dir = os.path.join(checkpoint_dir, config_name)
69     os.makedirs(config_checkpoint_dir, exist_ok=True)
70
71     # Salva configurazione
72     config_path = os.path.join(config_checkpoint_dir, 'config.json')
73     with open(config_path, 'w') as f:
74         json.dump(config, f, indent=4)
75
76     # Training history
77     history = {
78         'train_loss': [],
79         'train_acc': [],
80         'val_loss': [],
81         'val_acc': []
82     }
83
84     best_val_acc = 0.0
85
86     print(f"TRAINING: {config_name}")
87     print(f"Model config: {model_config}")
88     print(f"Training config: {training_config}")
89

```

```

90     num_epochs = training_config['num_epochs']
91
92     for epoch in range(num_epochs):
93         print(f"\nEpoch {epoch+1}/{num_epochs}")
94         print("-" * 70)
95
96         # Training, utilizzo funzione di training
97         train_loss, train_acc = train_one_epoch(
98             model, train_loader, criterion, optimizer, device
99         )
100
101        # Validation
102        val_loss, val_acc = validate(
103            model, test_loader, criterion, device
104        )
105
106        # Update scheduler
107        scheduler.step(val_acc)
108
109        # Salva storia
110        history['train_loss'].append(train_loss)
111        history['train_acc'].append(train_acc)
112        history['val_loss'].append(val_loss)
113        history['val_acc'].append(val_acc)
114
115        # Print summary
116        print(f"\nEpoch {epoch+1} Summary:")
117        print(f"  Train Loss: {train_loss:.4f} | Train Acc: {train_acc:.2f}%")
118        print(f"  Val Loss: {val_loss:.4f} | Val Acc: {val_acc:.2f}%")
119
120        # Salva best model
121        if val_acc > best_val_acc:
122            best_val_acc = val_acc
123            best_model_path = os.path.join(config_checkpoint_dir, 'best_model.pth')
124            save_checkpoint(model, optimizer, config, epoch+1, val_acc, best_model_path)
125            print(f" *** Nuovo best model! Accuracy: {val_acc:.2f}% ***")
126
127        # Early stopping check
128        if early_stopping(val_acc):
129            print(f"\nEarly stopping at epoch {epoch+1}")
130            break
131
132    print(f"TRAINING COMPLETATO: {config_name}")
133    print(f"Best Validation Accuracy: {best_val_acc:.2f}%")
134
135    return model, history, best_val_acc

```

Listing 15: Funzione per fare il training su una singola configurazione

```

1 def train_one_epoch(model, train_loader, criterion, optimizer,
2                     device):
3     """
4     Training per una singola epoca
5
6     Args:

```

```

6     model: modello da addestrare
7     train_loader: DataLoader training set
8     criterion: loss function
9     optimizer: ottimizzatore
10    device: device (cuda/cpu)
11
12    Returns:
13        avg_loss, accuracy
14    """
15
16    model.train()
17    running_loss = 0.0
18    correct = 0
19    total = 0
20
21    pbar = tqdm(train_loader, desc="Training")
22    for images, labels in pbar:
23        images, labels = images.to(device), labels.to(device)
24
25        # Forward pass
26        optimizer.zero_grad()
27        outputs = model(images)
28        loss = criterion(outputs, labels)
29
30        # Backward pass
31        loss.backward()
32        optimizer.step()
33
34        # Statistiche
35        running_loss += loss.item()
36        _, predicted = outputs.max(1)
37        total += labels.size(0)
38        correct += predicted.eq(labels).sum().item()
39
40        # Update progress bar
41        pbar.set_postfix({
42            'loss': f'{loss.item():.4f}',
43            'acc': f'{100.*correct/total:.2f}%'})
44
45    avg_loss = running_loss / len(train_loader)
46    accuracy = 100. * correct / total
47
48    return avg_loss, accuracy
49
50
51 def validate(model, test_loader, criterion, device):
52     """
53     Validazione sul test set
54
55     Args:
56         model: modello da validare
57         test_loader: DataLoader test set
58         criterion: loss function
59         device: device (cuda/cpu)
60
61     Returns:
62         avg_loss, accuracy
63     """

```

```
64 model.eval()
65 running_loss = 0.0
66 correct = 0
67 total = 0
68
69 with torch.no_grad():
70     pbar = tqdm(test_loader, desc="Validation")
71     for images, labels in pbar:
72         images, labels = images.to(device), labels.to(device)
73
74         outputs = model(images)
75         loss = criterion(outputs, labels)
76
77         running_loss += loss.item()
78         _, predicted = outputs.max(1)
79         total += labels.size(0)
80         correct += predicted.eq(labels).sum().item()
81
82         pbar.set_postfix({
83             'loss': f'{loss.item():.4f}',
84             'acc': f'{100.*correct/total:.2f}%',
85         })
86
87 avg_loss = running_loss / len(test_loader)
88 accuracy = 100. * correct / total
89
90 return avg_loss, accuracy
```

Listing 16: Funzioni di supporto al training

8 Confronto Finale tra le Architetture

Per fornire una visione d'insieme delle prestazioni ottenute, è stato effettuato un confronto diretto tra le architetture implementate (solo le migliori per ogni tipo di modello) su immagini di un dataset personale e uno con immagini generate con l'intelligenza artificiale generativa (Modelli: Nano-Banana e Chat-Gpt-5)

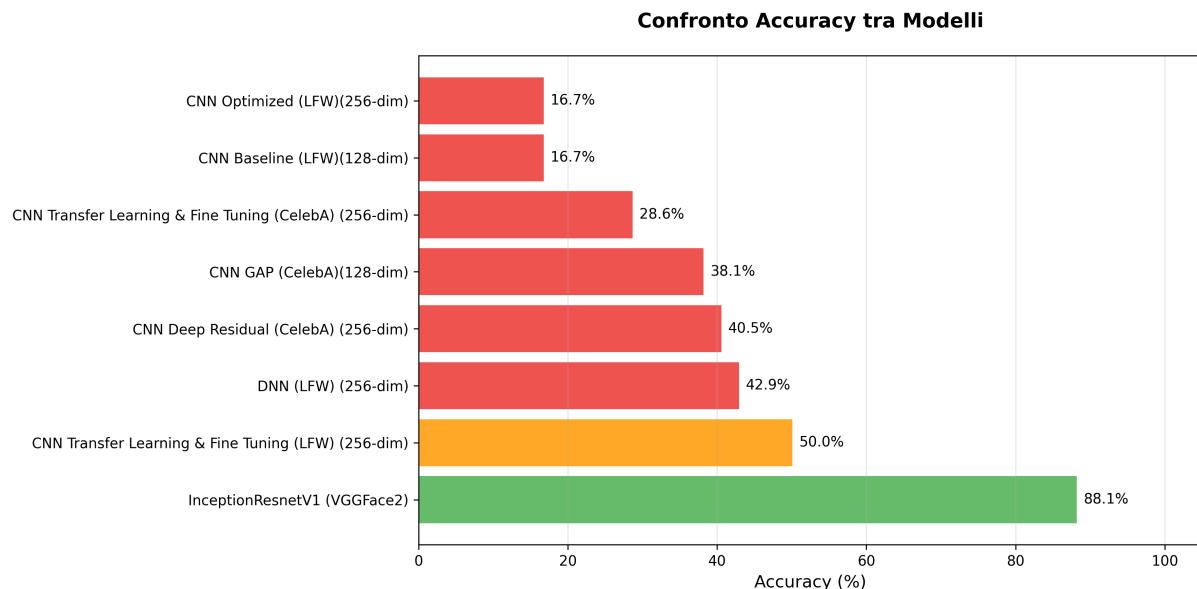


Figura 18: Accuracy delle reti utilizzate nell'applicazione su immagini di persone

Figura 19: Risultati delle reti utilizzate nell'applicazione

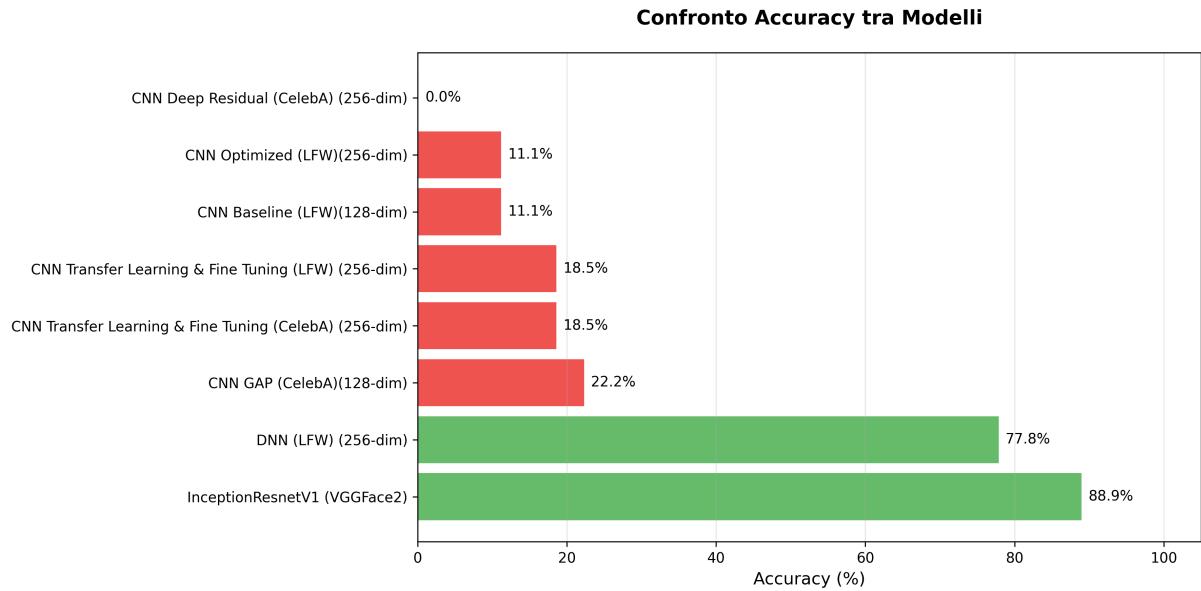


Figura 20: Accuracy delle reti utilizzate nell'applicazione su immagini di persone GENERATE con intelligenza artificiale generativa

Figura 21: Risultati delle reti utilizzate nell'applicazione su immagini GENERATE con intelligenza artificiale generativa