Relazione progetto di Laboratorio di Ottimizzazione, Intelligenza Artificiale e Machine Learning

A cura di Cacchi Alessandro

Matricola: 001080687

E-mail istituzionale: alessandro.cacchi@studio.unibo.it

Il progetto presentato è lo sviluppo di un'**architettura di rete neurale** volta a risolvere un problema di **classificazione multi-classe**. Utilizzo modelli pre-addestrati (*ResNet50* e *EfficientNet*) e tecniche di **preprocessing** per migliorare le performance.

analyze_dataset.py

In questo caso è stato creato un sistema di classificazione delle immagini in quattro categorie principali:

- **buildings** → edifici e palazzi
- **mountain** → paesaggi di montagna
- **sea** → paesaggi di mare
- **street** → strade/vie anche residenziali

Bilanciamento del dataset: Ho fatto un controllo preliminare per verificare il bilanciamento tra le classi, per evitare che il modello si concentri su una categoria a scapito delle altre.

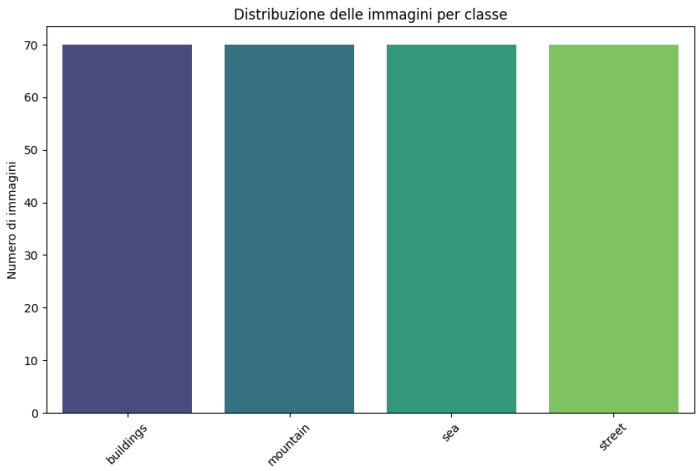
```python
def analyze_dataset(labels_df):

    # Conta il numero di immagini per ogni classe
    class_counts = labels_df['class'].value_counts()

    print("Distribuzione delle classi:")
    print(class_counts)

    plt.figure(figsize=(10, 6))
    sns.barplot(x=class_counts.index, y=class_counts.values, palette="viridis")
    plt.xlabel("Classi")
    plt.ylabel("Numero di immagini")
    plt.title("Distribuzione delle immagini per classe")
    plt.xticks(rotation=45)
    plt.show()
```

In questo caso, come si può notare dal grafico a barre prodotto, il dataset è perfettamente bilanciato:



dataset.py

Con questo file sono andato a gestire il caricamento e la preparazione del dataset per il training e la validazione della rete neurale.

Nella classe **PlaceDataset** sono andato a sviluppare tutte le varie funzioni:

```
def __init__(self, images_dir, labels_df, transform=None):
    self.images_dir = images_dir
    self.labels = labels_df
    self.transform = transform
```

Qui vado ad inizializzare i tre componenti principali: **images_dir** (directory delle immagini), **labels_df** (dataframe con le informazioni sulle immagini e le loro etichette e **transform** (trasformazioni da applicare alle immagini)

```
def __getitem__(self, idx):
    row = self.labels.iloc[idx]
    img_path = os.path.join(self.images_dir, row['filename'])
    image = Image.open(img_path).convert("RGB")
    label = row['class']
    if self.transform:
        image = self.transform(image)
    # Converte la label in tensore numerico utilizzando class_to_idx
    label_idx = self.class_to_idx[label]
    label_tensor = torch.tensor(label_idx, dtype=torch.long)
    return image, label_tensor
```

Successivamente abbiamo la funzione *_getitem_* che serve a fornire un singolo campione del dataset, ovvero una coppia chiave/valore.

Qui viene recuperata la riga corrispondente da labels_df, costruito il path dell'immagine (*img_path*), caricata l'immagine e convertita in RGB (*image*), applicate eventuali trasformazioni all'immagine, convertita l'etichetta in un tensore numerico usando il dizionario *class_to_indx* ì. Infine restituisce una coppia: *image*, ovvero l'immagine trasformata e *label_tensor*, il tensore corrispondente all'etichetta

```
def create_labels(image_dir, classes):
    data = []
    for class_idx, class_name in enumerate(classes):
        class_folder = os.path.join(image_dir, class_name)
        for filename in os.listdir(class_folder):
            if filename.endswith(".jpg"):
                data.append({"filename": os.path.join(class_name, filename), "class": class_name})
    return pd.DataFrame(data)
```

Costruisco un DataFrame contenente le informazioni sulle immagini: scorre la directory delle immagini, cerca il file .jpg e crea una tabella con due colonne (*filename* e *class*).

data_trasforms.py

Preprocessing delle immagini: Ho preprocessato le immagini tramite le trasformazioni fornite dalla libreria **torchvision**.

Ho anche applicato tecniche di **data augmentation** per migliorare le capacità del modello di generalizzare su immagini nuove e non viste.

```python
def get_transforms():
    train_transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.RandomHorizontalFlip(p=0.5),  # Flipping casuale
        transforms.RandomRotation(degrees=15),  # Rotazione casuale
        transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])

    test_transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])

    return train_transform, test_transform
```

Tecniche utilizzate:

- **Ridimensionamento (*Resize*):**
  - ho scalato tutte le immagini a una dimensione di 224x224 pixel.
- **Flipping orizzontale (*RandomHorizonalFlip*):**
  - Viene applicato con una probabilità del **50%** per ribaltare orizzontalmente le immagini.
- **Rotazione casuale (*RandomRotation*):**
  - Introduce rotazione casuale con un angolo massimo di 15 gradi.
- **Variazioni di colore (*ColorJitter*):**
  - Introduce variazioni casuali di **luminosità**, **contrasto**, **saturazione** e **tonalità**.
- **Normalizzazione dei pixel (*Normalize*):**
  - ho scalato i valori dei pixel usando le medie e deviazioni standard di ImageNet, per allinearsi alle reti preaddestrate.

```python
def split_data(labels, val_size, test_size, random_state): #accetta direttamente parametri di configurazione tramite test_size
    train_val_labels, test_labels = train_test_split(
        labels, test_size=test_size, stratify=labels['class'], random_state=random_state
    )
    train_labels, val_labels = train_test_split(
        train_val_labels, test_size=val_size / (1 - test_size), stratify=train_val_labels['class'], random_state=random_state
    )
    return train_labels, val_labels, test_labels
```

- **Test Split**: La funzione utilizza train_test_split di scikit-learn per separare una porzione di dati per il test set (test_size). La suddivisione è stratificata rispetto alla classe delle etichette - *stratify=labels['class']* - per garantire che la distribuzione delle classi rimanga uniforme.
- **Train/Validation Split**: Divide il rimanente set di dati (train/validation) in due parti, secondo la proporzione specificata da - *val_size / (1 - test_size)* –

```python
def create_dataloaders(image_dir, train_labels, val_labels, test_labels, train_transform, test_transform, batch_size):
    train_dataset = PlaceDataset(images_dir=image_dir, labels_df=train_labels, transform=train_transform)
    val_dataset = PlaceDataset(images_dir=image_dir, labels_df=val_labels, transform=test_transform)
    test_dataset = PlaceDataset(images_dir=image_dir, labels_df=test_labels, transform=test_transform)

    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    return train_loader, val_loader, test_loader, train_dataset, test_dataset
```

- **Dataset Creation**:
    - Crea tre oggetti PlaceDataset (dataset personalizzato) per training, validation e test, applicando le trasformazioni appropriate.
- **DataLoader Creation**:
    - Crea un DataLoader per ciascun dataset con - **batch_size** – (numero di campioni per batch) e – **shuffle** – (mescola i dati durante il training, ma non durante la validazione o il test).

training.py

Nel progetto ho implementato e confrontato due modelli di deep learning: **ResNet-50** e **EfficientNet**, entrambi preaddestrati su ImageNet.

Nel training dei modelli ho deciso di inserire una tecnica di **early stopping** per interrompere il training nel momento in cui il modello non mostra più miglioramenti sulla perdita di validazione per un numero consecutivo di epoche definito dal parametro **patience**.

Quindi, dopo che sono stati calcolati la perdita e l'accuratezza sul dataset di validazione, viene controllato se la perdita di validazione dell'epoca corrente (val_loss) è migliorata rispetto alla migliore perdita registrata finora (best_loss).

```
    if val_loss < best_loss:
        best_loss = val_loss
        patience_counter = 0
        best_model_state = model.state_dict()  # Salva lo stato del miglior modello
        torch.save(best_model_state, best_model_path)  # Salva su disco il miglior modello
        print(f"Salvato miglior modello: {best_model_path}")
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print("Early stopping triggered.")
            break
```

Risultati del training dei due modelli

ResNet 50:

Epoch 1/50, Train Loss: 1.0851, Train Acc: 58.33%, Val Loss: 0.7460, Val Acc: 75.00%

Salvato miglior modello: ResNet50_best.pth

Epoch 2/50, Train Loss: 0.4653, Train Acc: 91.67%, Val Loss: 0.4087, Val Acc: 83.93%

Salvato miglior modello: ResNet50_best.pth

Epoch 3/50, Train Loss: 0.2026, Train Acc: 95.24%, Val Loss: 0.3307, Val Acc: 87.50%

Salvato miglior modello: ResNet50_best.pth

.....Epoch dalla 4 alla 8.....

Epoch 9/50, Train Loss: 0.1015, Train Acc: 97.62%, Val Loss: 0.2933, **Val Acc: 94.64%**

Early stopping triggered.

Salvato ultimo modello: ResNet50_last.pth

EfficientNet:

Epoch 1/50, Train Loss: 1.3132, Train Acc: 38.69%, Val Loss: 1.2621, Val Acc: 46.43%

Salvato miglior modello: EfficientNet_best.pth

Epoch 2/50, Train Loss: 1.1466, Train Acc: 63.69%, Val Loss: 1.1130, Val Acc: 75.00%

Salvato miglior modello: EfficientNet_best.pth

Epoch 3/50, Train Loss: 1.0024, Train Acc: 80.36%, Val Loss: 0.9931, Val Acc: 76.79%

Salvato miglior modello: EfficientNet_best.pth

.....Epoch dalla 4 alla 32.....

Epoch 33/50, Train Loss: 0.1048, Train Acc: 98.81%, Val Loss: 0.2538, Val Acc: 91.07%

Salvato miglior modello: EfficientNet_best.pth

.....Epoch dalla 34 alla 37.....

Epoch 38/50, Train Loss: 0.0318, Train Acc: 100.00%, Val Loss: 0.2640, **Val Acc: 92.86%**

Early stopping triggered.

Salvato ultimo modello: EfficientNet_last.pth

Come possiamo vedere

- **ResNet50:** Dopo un rapido miglioramento iniziale, la **Val Loss** si stabilizza intorno a 0.2674 (epoca 4), ma successivamente aumenta leggermente. Questo porta a un early stopping all'epoca 9, quando la **Val Loss** è 0.2933 e la **Val Acc** è 94.64%. La stabilità del training è buona, ma l'early stopping viene attivato rapidamente poiché il modello non migliora ulteriormente.
- **EfficientNet:** Il training è più lungo e la **Val Loss** continua a migliorare fino all'epoca 33, quando raggiunge 0.2538. Il modello mostra una maggiore stabilità nel mantenere alte performance nelle epoche successive, anche se non migliora sempre in modo consistente. L'early stopping viene attivato all'epoca 38, indicando che il modello ha avuto più tempo per perfezionarsi.

Ho anche configurato il file **config.json** con vari parametri, tra questi:

- **batch_size** determina il numero di campioni elaborati in ciascun passaggio durante il training.
- Il numero di **epochs** stabilisce il massimo numero di iterazioni sull'intero dataset, ma l'**early stopping** garantisce che non tutte le epoche vengano necessariamente utilizzate, fermando il training quando non ci sono miglioramenti significativi.
- **learning_rate** controlla la velocità con cui il modello aggiorna i suoi pesi durante l'ottimizzazione.
- **patience** indica il numero massimo di epoche consecutive senza miglioramenti sulla perdita di validazione prima di attivare l'early stopping e interrompere il training.
- Per regolare dinamicamente il learning rate, vengono utilizzati **step_size** e **gamma**.
- La divisione del dataset, controllata da **test_size** e **val_size**, è fondamentale per valutare le performance del modello. La suddivisione in *training, test e validazione* garantisce che il modello venga addestrato su un insieme di dati e valutato su un altro, preservando la capacità di generalizzazione.
- **random_state** garantisce che i risultati siano riproducibili, mantenendo costanti le suddivisioni del dataset e il comportamento di altre operazioni randomiche.

```json
{
    "image_dir": "archive 4/images",
    "classes": ["buildings", "mountain", "sea", "street"],
    "batch_size": 32,
    "epochs": 50,
    "learning_rate": 0.0001,
    "step_size": 5,
    "gamma": 0.5,
    "patience": 5,
    "random_state": 42,
    "test_size": 0.2,
    "val_size": 0.2
}
```

Oltre al **config.json** ho configurato anche il **config_schema.json** che descrive appunto lo schema del primo file di configurazione (config.json).

```json
{
  "type": "object",
  "properties": {
    "image_dir": {
      "type": "string",
      "description": "Path alla directory che contiene le immagini del dataset.",
      "minLength": 1
    },
    "classes": {
      "type": "array",
      "description": "Elenco delle classi del dataset.",
      "items": {
        "type": "string",
        "minLength": 1
      },
      "minItems": 1
    },
    "batch_size": {
      "type": "integer",
      "description": "Numero di immagini per batch.",
      "minimum": 1
    },
    "epochs": {
      "type": "integer",
      "description": "Numero massimo di epoche di training.",
      "minimum": 1
    },
    "learning_rate": {
      "type": "number",
      "description": "Tasso di apprendimento per l'ottimizzatore.",
      "exclusiveMinimum": 0
    },
```

evaluation.py

Tramite sklearn ho generato il report con la funzione *classification_report*.

ResNet

	Precision	recall	F1-score	support
Sea	1.00	0.79	0.88	14
Street	0.76	0.93	0.84	14
Buildings	0.92	0.86	0.89	14
Mountain	0.93	1.00	0.97	14
Accuracy			0.89	56
Macro avg	0.91	0.89	0.89	56
Weighted avg	0.91	0.89	0.89	56

EfficientNet

	Precision	recall	F1-score	support
Sea	1.0	0.93	0.96	14
Street	0.92	0.86	0.89	14
Buildings	0.87	0.93	0.90	14
Mountain	0.93	1.00	0.97	14
Accuracy			0.93	56
Macro avg	0.93	0.93	0.93	56
Weighted avg	0.93	0.93	0.93	56

- **Precision:**

In entrambi i casi nella classe “sea”, la precisione è 1.00, il che significa che tutte le immagini predette come "sea" erano effettivamente corrette.

- **Recall:**

Nel caso di ResNet, nella classe "sea", il recall è 0.79, il che significa che il modello ha trovato il 79% di tutte le immagini effettivamente appartenenti alla classe "sea";

Mentre con EfficientNet si ha un recall di 0,86, leggermente incrementato rispetto a ResNet

- **F1-score:**

Nel caso di ResNet la classe "sea" ha un F1-score di 0.88, che è una misura bilanciata delle sue performance.

Mentre con EfficientNet si ha un F1-score di 0,96, leggermente incrementato rispetto a ResNet

- **Support:**

tutte le classi hanno un support di 14, il che significa che ci sono 14 immagini per ciascuna classe nel set di test.

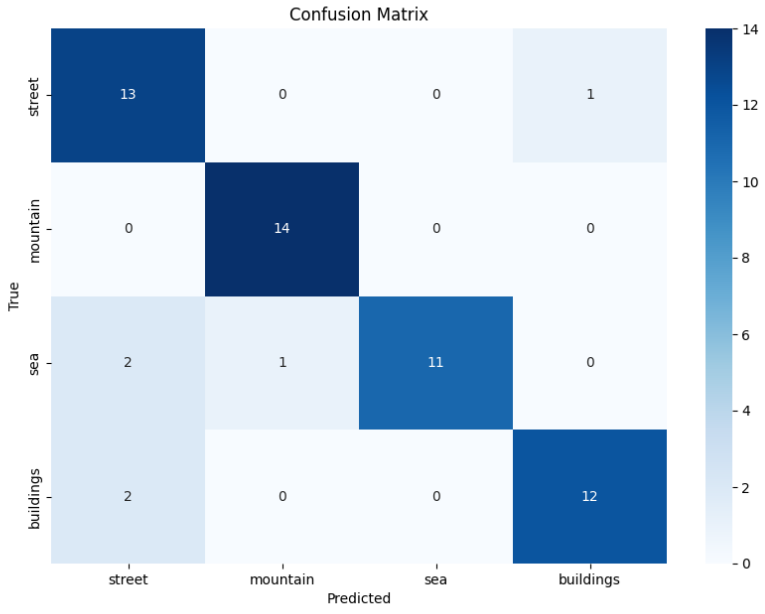
- **Macro avg**: EfficientNet ottiene una macro-media più alta in tutte le metriche (0.93 contro 0.91 per precision, e 0.93 contro 0.89 per recall e F1-score). Questo indica che EfficientNet è più consistente nelle prestazioni tra tutte le classi.
- **Weighted avg**: Anche in termini di media ponderata, EfficientNet ottiene punteggi superiori (0.93 contro 0.89), confermando che gestisce meglio la distribuzione dei dati nel dataset.

EfficientNet dimostra una maggiore capacità di generalizzazione rispetto a **ResNet50**, specialmente nelle classi con maggiore difficoltà come "Sea" e "Buildings". Sebbene ResNet abbia mostrato buone prestazioni complessive, EfficientNet risulta più accurato.

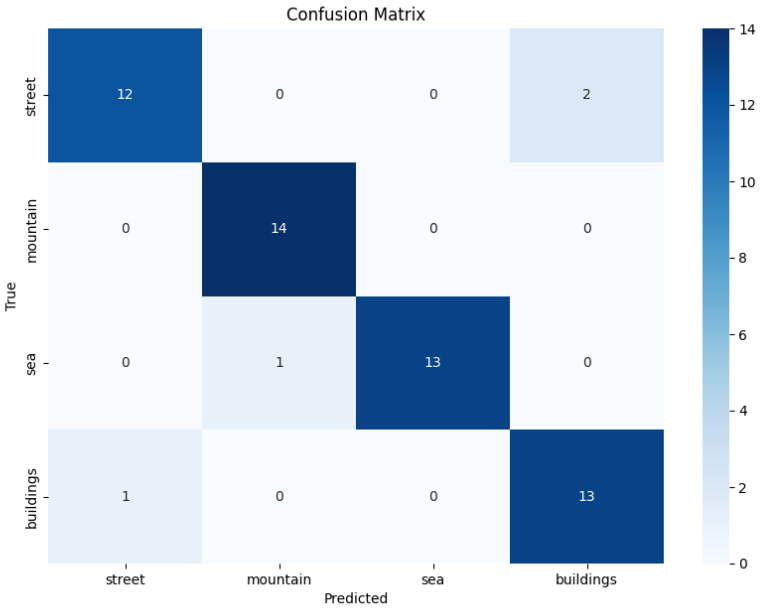
Confusion Matrix

Un'altra valutazione che ho svolto è stata quella di calcolare la *confusion matrix* tramite la funzione *confusion_matrix* sempre di **sklearn**. Viene generata una **heatmap** con le predizioni del modello sull'asse delle x e le etichette reali sull'asse delle y. Con essa posso identificare le classi più confuse del modello, evidenziando errori frequenti

ResNet:



EfficientNet:



Come si può vedere **EfficientNet** mostra un miglioramento rispetto a ResNet50, specialmente per la classe **sea**, con una riduzione degli errori (da 3 errori a 1).

La classe **mountain** viene classificata perfettamente con entrambi i modelli.

ResNet50 ottiene presstazioni leggermente migliori per la classe street, con un solo errore contro i 2 di EfficientNet.

EfficientNet sembra avere un vantaggio in termini di precisione complessiva, specialmente per la classe "sea" e "buildings".

immagini dimostrative delle predizioni

main.py

Il codice che gestisce l'addestramento e la valutazione dei due modelli è "diviso" in più parti. Iniziando con il caricamento e la validazione delle configurazioni:

```
# Carico lo schema di validazione
with open("config_schema.json", "r") as schema_file:
    config_schema = json.load(schema_file)

# Carico configurazioni da config.json
with open("config.json", "r") as config_file:
    config = json.load(config_file)

# Validazione configurazione
try:
    validate(instance=config, schema=config_schema)
    print("Configurazione valida.")
except ValidationError as e:
    print("Errore di validazione nella configurazione:", e)
    exit(1)
```

- Ho utilizzato file di configurazione (**config.json**) e uno schema di validazione (**config_schema.json**) per centralizzare i parametri.
- La validazione tramite **config_schema.json** assicura che i parametri rispettino le specifiche attese.

Successivamente ho impostato un **seed** per la riproducibilità, così da evitare risultati casuali che avrebbero potuto confondere l'analisi

```
# seed per la invariare comport. training
seed = config["random_state"]
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)  # Per CUDA
np.random.seed(seed)
random.seed(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

- **Trasformazioni:** Definisce trasformazioni per dati di addestramento e test.
- **Suddivisione:** Divide il dataset in training, validation e test.
- **Dataloader:** Crea i dataloader per iterare sui dati.

Le trasformazioni (get_transforms) standardizzano i dati, migliorando l'apprendimento del modello. La suddivisione (split_data) separa i dati in modo controllato per evitare contaminazioni tra training e test.

```python
# Trasformazioni e dataloader
train_transform, test_transform = get_transforms()
train_labels, val_labels, test_labels = split_data(
    labels,
    val_size=config["val_size"],
    test_size=config["test_size"],
    random_state=config["random_state"]
)

train_loader, val_loader, test_loader, train_dataset, test_dataset = create_dataloaders(
    image_dir,
    train_labels,
    val_labels,
    test_labels,
    train_transform,
    test_transform,
    batch_size=config["batch_size"]
)
```

Configurazione di perdita, ottimizzatori e scheduler

```python
# Configurazione perdita e ottimizzatore
criterion = nn.CrossEntropyLoss()

# Configurazione ottimizzatore e scheduler
resnet_optimizer = optim.Adam(resnet_model.parameters(), lr=config["learning_rate"])
efficientnet_optimizer = optim.Adam(efficientnet_model.parameters(), lr=config["learning_rate"])

resnet_scheduler = StepLR(resnet_optimizer, step_size=config["step_size"], gamma=config["gamma"])
efficientnet_scheduler = StepLR(efficientnet_optimizer, step_size=config["step_size"], gamma=config["gamma"])
```

nn.CrossEntropyLoss: È una funzione di perdita. Dice al modello quanto i suoi output si discostano dai valori reali, guidando l'ottimizzatore nella direzione giusta per migliorare le predizioni.

optim.Adam: L'ottimizzatore Adam (Adaptive Moment Estimation) è una versione avanzata della discesa del gradiente

- **resnet_model.parameters():** Specifica che i parametri del modello devono essere aggiornati.
- **lr=config["learning_rate"]:** Il learning rate (tasso di apprendimento) definisce quanto grandi devono essere gli aggiornamenti dei parametri.

Utilizzando gli ottimizzatori aggiorno i pesi del modello per ridurre la funzione di perdita.

Poi utilizzo anche uno **scheduler** per evitare che un *learning rate troppo alto* faccia oscillare il modello senza convergere mentre un *learning rate troppo basso* rallenti l'apprendimento

StepLR: È un tipo di scheduler che regola il learning rate in modo graduale dopo un numero fisso di epoche.

- **step_size:** Dopo quante epoche ridurre il learning rate.
- **gamma:** Fattore di riduzione. In questo caso, gamma=0.5, il learning rate viene moltiplicato per 0.5

Quindi:

- **Funzione di perdita (criterion):** Calcola l'errore del modello.
- **Ottimizzatori (resnet_optimizer e efficientnet_optimizer):** Utilizzano il gradiente della perdita per aggiornare i pesi del modello.
- **Scheduler (resnet_scheduler e efficientnet_scheduler):** Modificano dinamicamente il learning rate per migliorare l'efficienza dell'ottimizzatore.

Addestramento

```
# Training aggiornato con scheduler
train_model(
    resnet_model,
    train_loader,
    val_loader,
    criterion,
    resnet_optimizer,
    device,
    model_name="ResNet50",
    epochs=config["epochs"],
    patience=config["patience"]
)
resnet_scheduler.step()

train_model(
    efficientnet_model,
    train_loader,
    val_loader,
    criterion,
    efficientnet_optimizer,
    device,
    model_name="EfficientNet",
    epochs=config["epochs"],
    patience=config["patience"]
)
efficientnet_scheduler.step()
```

- **Training:** Addestro entrambi i modelli utilizzando il dataloader, la funzione di perdita e gli ottimizzatori.
- **Scheduler:** Aggiorno il learning rate dopo ogni epoca.
- Il parametro **patience=config["patience"]** serve per il meccanismo di **early stopping**, cioè interrompere l'addestramento se le performance sul validation set non migliorano per un certo numero di epoche consecutive

---

Infine, abbiamo la **valutazione** e la **visualizzazione**

```
# Valutazione e visualizzazione
evaluate_model(resnet_model, test_loader, train_dataset.get_idx_to_class(), device, model_name="ResNet50")
evaluate_model(efficientnet_model, test_loader, train_dataset.get_idx_to_class(), device, model_name="EfficientNet")

# Visualizzazione delle predizioni --> ResNet50
visualize_predictions(resnet_model, test_loader, train_dataset.get_idx_to_class(), device, model_name="ResNet50")

# Visualizzazione delle predizioni --> EfficientNet
visualize_predictions(efficientnet_model, test_loader, train_dataset.get_idx_to_class(), device, model_name="EfficientNet")
```

- **Valutazione:** Valuta le performance dei modelli sul test set.
- **Visualizzazione:** Mostra predizioni dei due modelli.