Relazione sul Codice di Preprocessing e Addestramento del Modello di Classificazione Di Tomografie Compiuterizzate TAC

1 Introduzione

Il codice fornito descrive il processo di addestramento di una rete neurale per la classificazione di immagini, con il modello pre-addestrato ResNet18, e include diverse tecniche di preprocessing per migliorare la qualità delle immagini. Il dataset è gestito tramite un DataFrame pandas, e vengono applicati vari metodi di preprocessing come la normalizzazione, l'augmentazione, e tecniche avanzate come Fuzzy C-Means, CLAHE, Wavelet e Retinex. Inoltre, il codice implementa la suddivisione dei dati in set di addestramento, validazione e test, e include la valutazione del modello utilizzando precision, recall, curva Precision-Recall, t-SNE, e UMAP.

2 Caricamento dei Dati

Il primo passo consiste nel caricare i dati dal file CSV contenente i percorsi delle immagini e le etichette. Il CSV viene caricato utilizzando la libreria `pandas`, e successivamente le etichette vengono mappate in valori numerici tramite un dizionario `label_map`:

```
names = ['image_path', 'organ', 'label']
df = pd.read_csv(r"..\Progetto\dataset.csv", names=names, header=0)

label_map = {
    "l.adenocarcinoma": 0,
    "l.benign": 1,
    "l.scadenocarcinoma": 2,
    "c.adenocarcinoma": 3,
    "c.benign": 4
}
df["label"] = df["label"].map(label_map)
```

Il DataFrame risultante conterrà i percorsi delle immagini e le etichette numeriche corrispondenti per ogni immagine.

3 Preprocessing delle Immagini

Sono implementati vari metodi di preprocessing delle immagini che vengono applicati a seconda delle esigenze. Le principali trasformazioni sono:

3.1 Standard Preprocessing

Il `standard_preprocessing` redimensiona l'immagine a 224x224 pixel, converte l'immagine in un tensor e la normalizza usando i valori medi e le deviazioni standard di ImageNet:

```
def standard_preprocessing():
    return transforms.Compose([
```

```
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
```

3.2 Augmentation Preprocessing

Il `augmentation_preprocessing` include operazioni di augmentation per migliorare la generalizzazione del modello, come il flip orizzontale, la rotazione casuale e la variazione del contrasto:

```
def augmentation_preprocessing():
    return transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(10),
        transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
```

3.3 Fuzzy C-Means Preprocessing

Il `fuzzy_preprocessing` applica l'algoritmo di clustering Fuzzy C-Means per segmentare l'immagine in tre cluster principali. Il risultato viene usato come una trasformazione dell'immagine:

```
[language=Python]
def fuzzy_preprocessing():
    def apply_fuzzy_cmeans(tensor_img):
        np_img = tensor_img.permute(1, 2, 0).numpy()
        gray = np.mean(np_img, axis=2)
        X = gray.flatten()
        X = np.expand_dims(X, axis=0)
        cntr, u, _, _, _, _, _ = fuzz.cluster.cmeans(X, c=3, m=2.0, error=0.005, maxiter=1
        cluster_map = np.argmax(u, axis=0).reshape(gray.shape)
        cluster_map = (cluster_map - cluster_map.min()) / (cluster_map.ptp() + 1e-6)
        cluster_tensor = torch.tensor(cluster_map, dtype=torch.float32)
        return cluster_tensor.unsqueeze(0).repeat(3, 1, 1)

    return transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Lambda(apply_fuzzy_cmeans),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
```

3.4 CLAHE Preprocessing

L'algoritmo `clahe_preprocessing` applica la tecnica CLAHE (Contrast Limited Adaptive Histogram Equalization) per migliorare il contrasto delle immagini:

```
def clahe_preprocessing():
    def apply_clahe(tensor_img):
        np_img = tensor_img.permute(1, 2, 0).numpy()
        np_img = (np_img * 255).astype(np.uint8)
        gray_img = cv2.cvtColor(np_img, cv2.COLOR_RGB2GRAY)
```

```
        clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
        clahe_img = clahe.apply(gray_img)
        clahe_img = clahe_img.astype(np.float32) / 255.0
        return torch.tensor(clahe_img).unsqueeze(0).repeat(3, 1, 1)

    return transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Lambda(apply_clahe),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
```

3.5 Wavelet Preprocessing

Il `wavelet_preprocessing` utilizza la trasformata wavelet per estrarre le caratteristiche salienti dell'immagine. La funzione `apply_wavelet` esegue la trasformata Haar:

```
def wavelet_preprocessing():
    def apply_wavelet(pil_img):
        img_np = np.array(pil_img.convert('L'))
        coeffs2 = pywt.dwt2(img_np, 'haar')
        cA, _ = coeffs2
        cA_img = Image.fromarray(cA).resize((224, 224))
        tensor = TF.to_tensor(cA_img).expand(3, -1, -1)
        return tensor

    return transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.Lambda(apply_wavelet),
        transforms.Normalize([0.5]*3, [0.5]*3)
    ])
```

3.6 Retinex Sobel Preprocessing

Il `retinex_sobel_preprocessing` applica un algoritmo Retinex per migliorare l'illuminazione e un filtro Sobel per il rilevamento dei bordi:

```
def retinex_sobel_preprocessing():
    def apply_retinex_sobel(pil_img):
        img_np = np.array(pil_img.convert('RGB'))
        img_gray = cv2.cvtColor(img_np, cv2.COLOR_RGB2GRAY)

        # Retinex algorithm
        retinex = np.log1p(img_gray) - np.log1p(cv2.GaussianBlur(img_gray, (5, 5), 0))

        # Sobel edge detection
        sobel_edges = cv2.Sobel(retinex, cv2.CV_64F, 1, 1, ksize=3)
        sobel_edges = cv2.convertScaleAbs(sobel_edges)

        # Normalize and return as tensor
        sobel_edges = sobel_edges.astype(np.float32) / 255.0
        return torch.tensor(sobel_edges).unsqueeze(0).repeat(3, 1, 1)

    return transforms.Compose([
```

```
        transforms.Resize((224, 224)),
        transforms.Lambda(apply_retinex_sobel),
        transforms.Normalize([0.5]*3, [0.5]*3)
    ])
```

4 Creazione del Dataset

Il dataset è creato utilizzando la classe personalizzata `CustomImageDataset`, che eredita dalla classe `Dataset` di PyTorch. Ogni immagine viene caricata tramite il percorso specificato nel DataFrame e viene applicata la trasformazione selezionata. La classe è definita come segue:

```
class CustomImageDataset(Dataset):
    def __init__(self, df, transform=None):
        self.df = df
        self.transform = transform

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        path = self.df.iloc[idx]['image_path']
        label = self.df.iloc[idx]['label']
        try:
            img = Image.open(path).convert("RGB")
            if self.transform:
                img = self.transform(img)
            return img, label
        except Exception as e:
            print(f"Errore immagine {path}: {e}")
            return torch.zeros(3, 224, 224), 0
```

5 Modello e Addestramento

Il modello utilizzato è una versione modificata di ResNet18, con l'ultimo layer completamente connesso (`fc`) adattato per 5 classi (a differenza delle 1000 classi di ImageNet). Il modello è addestrato utilizzando la funzione di perdita `CrossEntropyLoss` e l'ottimizzatore `Adam`. Il codice per il training del modello è il seguente:

```
model = models.resnet18(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, 5)  # Modifica l'output per 5 classi

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
```

Il modello è addestrato per un numero predefinito di epoche, e durante ogni epoca vengono calcolati la perdita e l'accuratezza sia sul training che sulla validazione. Il training è gestito dalla funzione `train_model`.

6 Valutazione del Modello

Una volta che il modello è addestrato, viene valutato sul set di test. Vengono calcolate e visualizzate le seguenti metriche:

- Accuratezza
- Matrice di Confusione
- Precision-Recall Curve
- UMAP e t-SNE per la visualizzazione delle embedding

Le matrici di confusione e le curve di precisione/recall sono utili per comprendere le performance del modello, mentre UMAP e t-SNE sono utilizzati per visualizzare le relazioni tra le predizioni.

7 Conclusioni

Il codice implementa un flusso completo di preprocessamento, addestramento, e valutazione di un modello di deep learning per la classificazione di immagini. Le diverse tecniche di preprocessing permettono di ottimizzare le immagini per l'addestramento del modello, mentre la valutazione fornisce strumenti per analizzare le performance del modello.