Explanation of ResNet 50 for Dataslayer

Salman Faiz Hidayat
Universitas Gadjah Mada

Tristan Khayru Abiyudha
Universitas Gadjah Mada

Adam Maulana Haq
Universitas Gadjah Mada

December 2024

1 Code

```
import os
import numpy as np
import pandas as pd
import torch
from torch.utils.data import DataLoader, Dataset
from torchvision.transforms import Compose, Resize, ToTensor, Normalize, RandomHorizontalFli
from torchvision import models
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, f1_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
from PIL import Image

# ---------------- STEP 1: Configuration ---------------- #
TRAIN_DIR = "/kaggle/input/dataslayer2/train"
TEST_DIR = "/kaggle/input/dataslayer2/test"
SUBMISSION_FILE = "submission.csv"
IMG_SIZE = (224, 224)
BATCH_SIZE = 16
EPOCHS = 20
LEARNING_RATE = 2e-5
RANDOM_STATE = 42

# ---------------- STEP 2: Data Preparation ---------------- #
```

```
def preprocess_dataset(directory):
    images, labels = [], []
    for subject in os.listdir(directory):
        subject_path = os.path.join(directory, subject)
        if not os.path.isdir(subject_path):
            continue
        for category in ["fall", "non_fall"]:
            category_path = os.path.join(subject_path, category)
            label = 1 if category == "fall" else 0
            if os.path.isdir(category_path):
                for root, _, files in os.walk(category_path):
                    for file in files:
                        if file.endswith((".jpg", ".png", ".jpeg")):
                            img_path = os.path.join(root, file)
                            images.append(img_path)
                            labels.append(label)
    return images, labels

X, y = preprocess_dataset(TRAIN_DIR)
X_train, X_val, y_train, y_val = train_test_split(
        X, y,
        test_size=0.2,
        stratify=y,
        random_state=RANDOM_STATE
    )
print(f"Training samples: {len(X_train)}, Validation samples: {len(X_val)}")

# ---------------- STEP 3: Dataset Definition ---------------- #
train_transform = Compose([
    Resize(IMG_SIZE),
    RandomHorizontalFlip(),
    RandomRotation(10),
    ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    ToTensor(),
    Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
])

val_transform = Compose([
    Resize(IMG_SIZE),
    ToTensor(),
    Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
])

class ImageDataset(Dataset):
    def __init__(self, image_paths, labels=None, transform=None):
        self.image_paths = image_paths
```

```
        self.labels = labels
        self.transform = transform

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

    def __getitem__(self, idx):
        image = Image.open(self.image_paths[idx]).convert("RGB")
        image = self.transform(image) if self.transform else image
        data = {"image": image}
        if self.labels is not None:
            data["label"] = torch.tensor(self.labels[idx], dtype=torch.long)
        return data

train_dataset = ImageDataset(X_train, y_train, transform=train_transform)
val_dataset = ImageDataset(X_val, y_val, transform=val_transform)
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE)

# ---------------- STEP 4: Model Definition ---------------- #
class CNN(nn.Module):
    def __init__(self, num_classes=2):
        super(CNN, self).__init__()
        self.model = models.resnet50(pretrained=True)
        self.model.fc = nn.Linear(self.model.fc.in_features, num_classes)

    def forward(self, x):
        return self.model(x)

model = CNN(num_classes=2).to(torch.device("cuda" if torch.cuda.is_available() else "cpu"))

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

# ---------------- STEP 5: Training Loop ---------------- #
def train_model(model, train_loader, val_loader, criterion, optimizer, epochs, device):
    train_losses, val_losses, val_f1_scores = [], [], []
    best_f1 = 0
    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        for batch in tqdm(train_loader):
            images = batch["image"].to(device)
            labels = batch["label"].to(device)

            optimizer.zero_grad()
```

```
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        train_loss = running_loss / len(train_loader)
        train_losses.append(train_loss)

        # Validation phase
        model.eval()
        val_loss, val_preds, val_labels = 0.0, [], []
        with torch.no_grad():
            for batch in val_loader:
                images = batch["image"].to(device)
                labels = batch["label"].to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                val_preds.extend(torch.argmax(outputs, dim=1).cpu().numpy())
                val_labels.extend(labels.cpu().numpy())

        val_loss /= len(val_loader)
        val_losses.append(val_loss)
        val_f1 = f1_score(val_labels, val_preds, average="weighted")
        val_f1_scores.append(val_f1)

        print(
                f"Epoch {epoch+1}/{epochs},
                Train Loss: {train_loss:.4f},
                Val Loss: {val_loss:.4f},
                Val F1: {val_f1:.4f}"
            )

        if val_f1 > best_f1:
            best_f1 = val_f1
            torch.save(model.state_dict(), "best_model.pth")
            print(f"Saved Best Model with F1: {best_f1:.4f}")

    return train_losses, val_losses, val_f1_scores

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
train_losses, val_losses, val_f1_scores = train_model(
    model,
    train_loader,
    val_loader,
```

```
    criterion,
    optimizer,
    EPOCHS,
    device
)

# ---------------- STEP 6: Learning Curve ---------------- #
plt.figure(figsize=(10, 6))
plt.plot(train_losses, label="Training Loss")
plt.plot(val_losses, label="Validation Loss")
plt.title("Learning Curve")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()

# ---------------- STEP 7: Validation Metrics ---------------- #
best_model = CNN(num_classes=2)
best_model.load_state_dict(torch.load("best_model.pth"))
best_model = best_model.to(device)
best_model.eval()

val_preds, val_labels, val_probs = [], [], []
with torch.no_grad():
    for batch in val_loader:
        images = batch["image"].to(device)
        labels = batch["label"].to(device)
        outputs = best_model(images)
        probs = torch.softmax(outputs, dim=1)[:, 1].cpu().numpy()
        val_preds.extend(torch.argmax(outputs, dim=1).cpu().numpy())
        val_probs.extend(probs)
        val_labels.extend(labels.cpu().numpy())

print("\nClassification Report:")
print(classification_report(val_labels, val_preds, target_names=["non_fall", "fall"]))

conf_matrix = confusion_matrix(val_labels, val_preds)
plt.figure(figsize=(10, 8))
sns.heatmap(
        conf_matrix,
        annot=True,
        fmt="d",
        cmap="Blues",
        xticklabels=["non_fall", "fall"],
        yticklabels=["non_fall", "fall"]
    )
```

```
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()

# ---------------- STEP 8: Test Predictions with Threshold Optimization ---------------- #
class ImageDataset(Dataset):
    def __init__(self, image_paths, labels=None, transform=None):
        self.image_paths = image_paths
        self.labels = labels
        self.transform = transform

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

    def __getitem__(self, idx):
        try:
            image = Image.open(self.image_paths[idx]).convert("RGB")
        except Exception as e:
            print(f"Error loading image: {self.image_paths[idx]}, {e}")
            image = Image.new("RGB", IMG_SIZE)  # Use a blank image as a fallback

        image = self.transform(image) if self.transform else image
        data = {"image": image, "path": self.image_paths[idx]}  # Store path for later use
        if self.labels is not None:
            data["label"] = torch.tensor(self.labels[idx], dtype=torch.long)
        return data

test_images = [
        os.path.join(TEST_DIR, img) for img in os.listdir(TEST_DIR)
        if img.endswith((".jpg", ".png", ".jpeg"))
    ]
test_dataset = ImageDataset(test_images, transform=val_transform)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE)

test_probs, test_ids = [], []
with torch.no_grad():
    for batch in test_loader:
        images = batch["image"].to(device)
        outputs = best_model(images)
        probs = torch.softmax(outputs, dim=1)[:, 1].cpu().numpy()
        test_probs.extend(probs)
        test_ids.extend([os.path.basename(path) for path in batch["path"]])

# Find optimal threshold using validation probabilities
if len(val_probs) > 0:  # Ensure val_probs is not empty
```

```
    best_threshold, best_f1 = 0, 0
    thresholds = np.linspace(0.1, 0.9, 50)
    for threshold in thresholds:
        val_preds_threshold = (np.array(val_probs) > threshold).astype(int)
        f1 = f1_score(val_labels, val_preds_threshold)
        if f1 > best_f1:
            best_f1, best_threshold = f1, threshold

    print(f"Best Threshold: {best_threshold:.2f}, Best F1 Score: {best_f1:.4f}")

    # Apply best threshold to test data
    test_preds = (np.array(test_probs) > best_threshold).astype(int)

    submission = pd.DataFrame({"id": test_ids, "label": test_preds})
    submission.to_csv(SUBMISSION_FILE, index=False)
    print(f"Submission saved with threshold {best_threshold:.2f}")
else:
    print("Validation probabilities are empty. Skipping threshold optimization.")
```

2 Explanation

We also explored the use of another model, ResNet50 [4]. ResNet50 was chosen because it is a well-established and powerful model for solving computer vision problems. ResNet50 works by adding the input features from earlier layers to the output features of layers deeper in the network. This is called a skip connection. Skip connections minimize or eliminate the negative impact of certain layers that might degrade model performance. The effect of skip connections is similar to regularization parameters in other models (like L1 or L2 regularization). This method allows for the addition of many layers without risking vanishing gradients, where weights shrink due to the large number of layers. Very small weights prevent the model from generalizing predictions on input images [1] [3] [5]. Figure 1 illustrates skip connections and the ResNet50 architecture [2]:

Another key feature of this implementation is the use of image data augmentation. Augmentation is applied to help the model generalize to input images with slight variations. This process generates new samples from the given dataset by introducing minor changes, such as mirroring or rotating images. Below is the implementation code for the augmentations:

```
train_transform = Compose([
    # Resize images for consistent training input
    Resize(IMG_SIZE),

    # Randomly flip images horizontally with a 50% probability
    RandomHorizontalFlip(),
```

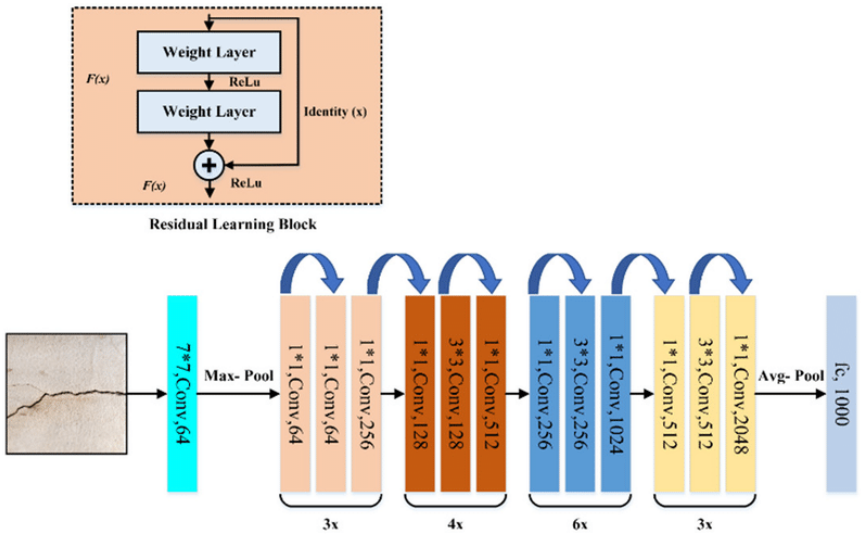

Figure 1: Illustration of Skip Connections and the ResNet50 Architecture

```
    # Randomly rotate images up to 10 degrees
    RandomRotation(10),

    # Modify brightness, contrast, saturation, and hue:
    # Brightness: Multiply pixel values by factors between 0.8 and 1.2
    # Contrast: Multiply pixel values relative to the mean intensity of the image.
    # Saturation: Adjust scaling of RGB values.*
    # Hue: Shift pixel colors by ±0.1 (normalized range).
    ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),

    # Convert images to numpy arrays and normalize pixel values
    ToTensor(),

    # Normalize pixel values for RGB channels
    Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
])

val_transform = Compose([
    Resize(IMG_SIZE),
    ToTensor(),
    Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
])
```

*For saturation adjustments, two steps are involved:

1. Convert RGB values to grayscale using the formula:

$$Gray = 0.2989 \times R + 0.5870 \times G + 0.1140 \times B$$

2. Adjust RGB values using the formula:

$$Adjusted\_RGB = Gray + (Original\_RGB - Gray) \times saturation\_factor$$

Explanation of each variable used in RGB adjustment:

- $Adjusted\_RGB$: The adjusted RGB values after saturation modification.
- $Original\_RGB$: The original RGB values of the pixel.
- $Gray$: The grayscale equivalent of the pixel, calculated from RGB:
  - 0.2989: Weight for the red channel.
  - 0.5870: Weight for the green channel (most sensitive to human vision).
  - 0.1140: Weight for the blue channel.
- $saturation\_factor$: The scaling factor for saturation adjustments:
  - $saturation\_factor = 1.0$: No change to saturation.
  - $saturation\_factor < 1.0$: Reduces saturation, closer to grayscale.
  - $saturation\_factor > 1.0$: Increases saturation, making colors more vibrant.

References

[1] Residual networks (resnet) - deep learning - geeksforgeeks, 1 2023.

[2] Understanding resnet-50 in depth: Architecture, skip connections, and advantages over other networks - wisdom ml, 3 2023.

[3] Ryan Ahmed. (532) what is resnet? (with 3d visualizations) - youtube, 10 2022.

[4] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2015.

[5] Rupert Meneer. (532) resnet (actually) explained in under 10 minutes - youtube, 3 2022.