NAME: BHOOMIKA NANJARAJA
GWID: G41609848

Generating FEN-notations from Chess Board Images

1. Introduction

Overview of the Project: The project focuses on developing a deep learning model capable of identifying and classifying positions on a chessboard and converting them into the standard **Forsyth-Edwards Notation (FEN)**. This is achieved using labeled chessboard images. The goal is to design a system that performs accurate predictions for static chessboard configurations and provides real-time detection capability.

Shared Work Overview: The team collaborated on various aspects of the project:

- **Dataset Preparation:** Selecting a Kaggle dataset with annotated chessboard images.
- **Model Development:** Evaluating and training models like AlexNet, EfficientNet, and Vision Transformers (ViT).
- **Object Detection:** Leveraging YOLO for real-time image detection and handling images with background noise.
- **Frontend Development:** Creating a user-friendly Streamlit-based interface for image uploads and FEN notation display.
- **Evaluation:** Assessing model performance using metrics such as accuracy, precision, recall, and F1 score.

My individual contribution focused on:

1. Developing a preprocessing pipeline for splitting chessboard images into 64 grids, resizing them to 224x224 pixels, and applying data augmentation for generalization.
2. Designing the training pipeline using AlexNet, incorporating Cross-Entropy Loss, Adam optimizer, early stopping, and learning rate scheduling.
3. Building a Streamlit application to allow users to upload chessboard images and obtain FEN notation predictions in real-time.

Algorithm:

1. **Preprocessing**:
    - Split chessboard image into 64 grids.
    - Resize each grid to 224x224 and apply data augmentation (rotation, contrast adjustment, etc.) to improve generalization.

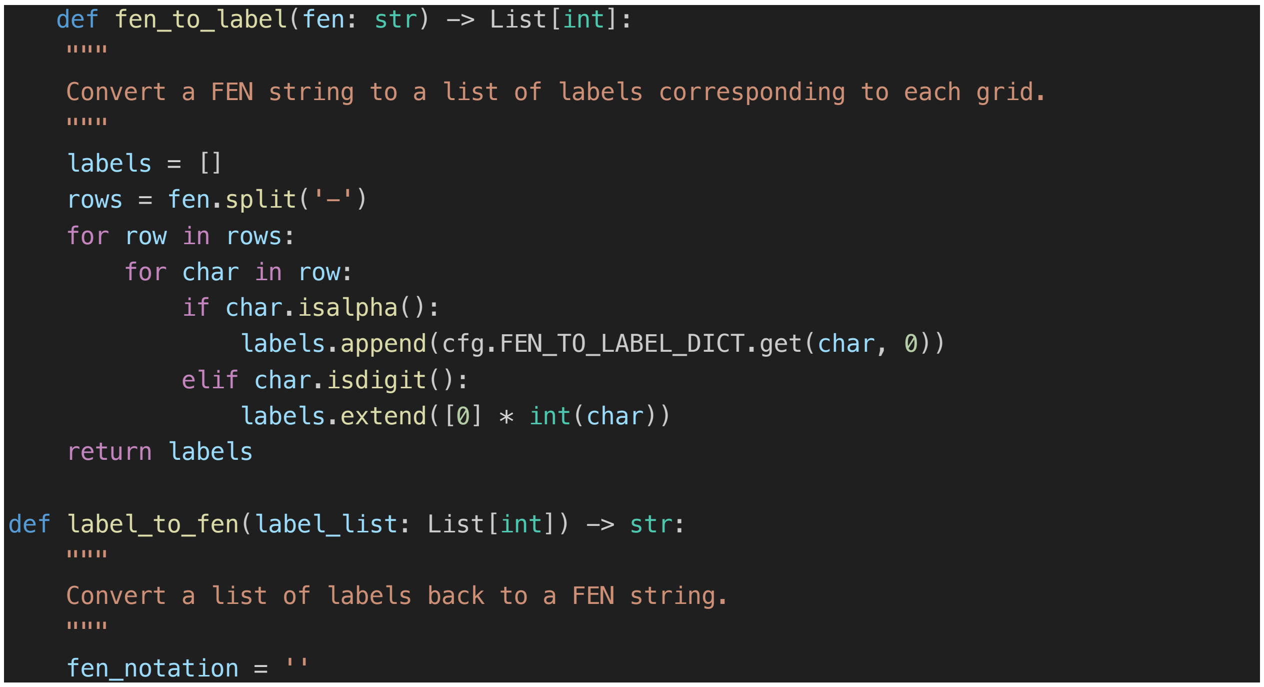
2. **Model Training**:
    - Use AlexNet with custom fully connected layers for 13 output classes.
    - Apply Cross-Entropy Loss and Adam optimizer with learning rate scheduling.
    - Implement early stopping to prevent overfitting.

3. **FEN Conversion**:
    - Map model predictions (labels) back to FEN notation using LABEL_TO_FEN_SHORT.

4. **Streamlit App**:
    - Upload chessboard images, predict the FEN notation, and display the result alongside the uploaded image.

Here's a detailed explanation of each function and how they helped in training the model, as well as why I am using specific image preprocessing techniques:

Preprocessing and Data Pipeline



```
    def fen_to_label(fen: str) -> List[int]:
     """
     Convert a FEN string to a list of labels corresponding to each grid.
     """
     labels = []
     rows = fen.split('-')
     for row in rows:
         for char in row:
             if char.isalpha():
                 labels.append(cfg.FEN_TO_LABEL_DICT.get(char, 0))
             elif char.isdigit():
                 labels.extend([0] * int(char))
     return labels

def label_to_fen(label_list: List[int]) -> str:
     """
     Convert a list of labels back to a FEN string.
     """
     fen_notation = ''
     empty_count = 0
     for idx, label in enumerate(label_list):
         if label == 0:
             empty_count += 1
         else:
             if empty_count > 0:
                 fen_notation += str(empty_count)
                 empty_count = 0
             fen_notation += cfg.LABEL_TO_FEN_SHORT.get(label, '0')
         if (idx + 1) % 8 == 0:
             if empty_count > 0:
                 fen_notation += str(empty_count)
                 empty_count = 0
             if idx != 63:
                 fen_notation += '-'
     return fen_notation

 def split_image_into_grids(image: np.ndarray, grid_size: Tuple[int, int] = (50, 50)) -
 > List[np.ndarray]:
     """
     Split the chessboard image into 64 grids.
     """
     grids = []
     h, w = image.shape[:2]
     grid_h, grid_w = grid_size
     for i in range(0, h, grid_h):
         for j in range(0, w, grid_w):
             grid = image[i:i + grid_h, j:j + grid_w]
             if grid.shape[0] == grid_h and grid.shape[1] == grid_w:
                 grids.append(grid)
     return grids



 class ChessDataset(Dataset):
     """
     Custom Dataset for Chess Piece Classification.
     Processes images on-the-fly without pre-processing.
     """

     def __init__(self, folder: str, transform=None):
         self.image_paths = self.get_image_paths(folder)
         self.transform = transform
         self.labels = self.prepare_labels()

     def get_image_paths(self, folder: str) -> List[str]:
         extensions = ['.jpeg', '.jpg', '.JPEG', '.JPG', '.png', '.bmp', '.gif']
         image_paths = []
```

```python
        for ext in extensions:
            image_paths.extend(glob.glob(os.path.join(folder, f"*{ext}")))
        return image_paths

    def prepare_labels(self) -> List[List[int]]:
        labels = []
        for path in self.image_paths:
            filename = os.path.splitext(os.path.basename(path))[0]
            labels.append(fen_to_label(filename))
        return labels

    def __len__(self):
        return len(self.image_paths) * 64  # 64 grids per image

    def __getitem__(self, idx):
        image_idx = idx // 64
        grid_idx = idx % 64
        img_path = self.image_paths[image_idx]
        label = self.labels[image_idx][grid_idx]

        # Load image
        image = cv2.imread(img_path)
        if image is None:
            # Handle missing images by using a default image
            print(f"Warning: Image {img_path} could not be read. Using blank grid.")
            image_rgb = np.zeros((cfg.IMAGE_SIZE, cfg.IMAGE_SIZE, cfg.CHANNELS),
dtype=np.uint8)
        else:
            image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
            grids = split_image_into_grids(image_gray, grid_size=(50, 50))
            if grid_idx >= len(grids):
                # Handle cases where grid_idx is out of bounds
                print(f"Warning: Grid index {grid_idx} out of bounds for image:
{img_path}. Using blank grid.")
                image_rgb = np.zeros((cfg.IMAGE_SIZE, cfg.IMAGE_SIZE, cfg.CHANNELS),
dtype=np.uint8)
            else:
                grid = grids[grid_idx]
                grid_resized = cv2.resize(grid, (cfg.IMAGE_SIZE, cfg.IMAGE_SIZE))
                image_rgb = cv2.cvtColor(grid_resized, cv2.COLOR_GRAY2RGB)

        pil_image = Image.fromarray(image_rgb)


        if self.transform:
            # Albumentations expects images in numpy array format
            transformed = self.transform(image=np.array(pil_image))
            image = transformed['image']
        else:
            transform = transforms.Compose([
                transforms.ToTensor(),
                transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                     std=[0.229, 0.224, 0.225])
            ])
            image = transform(pil_image)

        # Convert label to tensor
        label = torch.tensor(label, dtype=torch.long)

        return image, label



train_transform = A.Compose([
    A.Resize(height=cfg.IMAGE_SIZE, width=cfg.IMAGE_SIZE),  # Ensure resizing
    A.HorizontalFlip(p=0.5),
    A.Rotate(limit=15, p=0.5),
    A.RandomBrightnessContrast(p=0.5),
    A.ShiftScaleRotate(shift_limit=0.05, scale_limit=0.05, rotate_limit=10, p=0.5),
    A.GridDistortion(p=0.3),
    A.ElasticTransform(p=0.3),
    A.CLAHE(p=0.5),
    A.Normalize(mean=(0.485, 0.456, 0.406),
                std=(0.229, 0.224, 0.225)),
    ToTensorV2()
])

val_transform = A.Compose([
    A.Resize(height=cfg.IMAGE_SIZE, width=cfg.IMAGE_SIZE),  # Ensure resizing
    A.Normalize(mean=(0.485, 0.456, 0.406),
                std=(0.229, 0.224, 0.225)),
    ToTensorV2()
])
```

1. **Image Splitting (split_image_into_grids)**:
    - **Functionality**: The chessboard image is split into 64 smaller grids (8x8) using the split_image_into_grids function. This step is essential because FEN notation maps each chessboard piece to a specific grid on the board. By splitting the image into grids, each grid can be treated as an individual input for the model.
    - **Impact**: This preprocessing step ensures that the model can learn to classify each individual piece and its position independently, which is necessary for accurate prediction of FEN notation.

2. **Image Resizing**:
   - **Functionality**: Each grid is resized to 224×224 pixels before being input to the model. This resizing is necessary because the deep learning models (like AlexNet) expect inputs of a consistent size.
   - **Why it's needed**: Models like AlexNet are pre-trained on images of specific sizes, so resizing ensures compatibility with these models. It also reduces computation time by ensuring each image is a manageable size.

3. **Normalization**:
   - **Functionality**: The images are normalized using a mean and standard deviation. The mean and standard deviation values used are the ones typical for ImageNet pre-trained models.
   - **Why it's needed**: Normalization ensures that the model receives inputs that are on a similar scale, which speeds up training and helps the model converge faster. Pre-trained models have typically been trained on normalized images, so applying the same transformation allows the model to generalize better to new data.

4. **Augmentations**:
   - **Functionality**: The train_transform includes a series of augmentations, such as:
     - **Rotation**: Helps the model learn invariance to the orientation of the pieces.
     - **Random Brightness and Contrast**: Helps the model generalize under different lighting conditions.
     - **Grid Distortion and Elastic Transform**: Makes the model robust to slight distortions or deformations in the chessboard.
   - **Why it's needed**: These augmentations prevent the model from overfitting and help it generalize to real-world variations (e.g., slight misalignments, lighting changes, etc)

5. **FEN_TO_LABEL_DICT**

```
FEN_TO_LABEL_DICT = {
        'p': 1, 'P': 2,
        'b': 3, 'B': 4,
        'r': 5, 'R': 6,
        'n': 7, 'N': 8,
        'q': 9, 'Q': 10,
        'k': 11, 'K': 12,
        '0': 0  # Representing empty grid}
```

Keys: FEN characters representing chess pieces.

- 'p': Black pawn
- 'P': White pawn
- 'b': Black bishop
- 'B': White bishop
- 'r': Black rook
- 'R': White rook
- 'n': Black knight
- 'N': White knight
- 'q': Black queen
- 'Q': White queen
- 'k': Black king
- 'K': White king
- '0': Empty square (represented by 0 for no piece).

Prepare_Dataloaders Function

```
    def prepare_dataloaders(folder: str, transform, batch_size: int =
cfg.BATCH_SIZE, shuffle: bool = True) -> DataLoader:
    dataset = ChessDataset(folder=folder, transform=transform)
    print(f"Loaded {len(dataset)} samples from {folder}")  # Debugging line
    if len(dataset) == 0:
        raise ValueError(f"No samples found in {folder}. Please check the
directory and file naming.")
    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=shuffle,
num_workers=cfg.NUM_WORKERS,
                            pin_memory=cfg.PIN_MEMORY,
prefetch_factor=cfg.PREFETCH_FACTOR,
                            persistent_workers=cfg.PERSISTENT_WORKERS)
    return dataloader

total_train_images = len(full_train_dataset.image_paths)
val_split = 0.2
val_size = int(total_train_images * val_split)
train_size = total_train_images - val_size

train_grid_size = train_size * 64
val_grid_size = val_size * 64

train_subset, val_subset = random_split(
    full_train_dataset,
    [train_grid_size, val_grid_size],
    generator=torch.Generator().manual_seed(42)
)

print(f"Number of training grids: {len(train_subset)}")
print(f"Number of validation grids: {len(val_subset)}")

train_loader = DataLoader(train_subset, batch_size=cfg.BATCH_SIZE,
shuffle=True, num_workers=cfg.NUM_WORKERS,
                            pin_memory=cfg.PIN_MEMORY,
prefetch_factor=cfg.PREFETCH_FACTOR,
                            persistent_workers=cfg.PERSISTENT_WORKERS)
val_loader = DataLoader(val_subset, batch_size=cfg.BATCH_SIZE, shuffle=False,
num_workers=cfg.NUM_WORKERS,
                        pin_memory=cfg.PIN_MEMORY,
prefetch_factor=cfg.PREFETCH_FACTOR,
                        persistent_workers=cfg.PERSISTENT_WORKERS)

test_loader = prepare_dataloaders(folder=cfg.TEST_DIR, transform=val_transform,
batch_size=cfg.BATCH_SIZE,
                                    shuffle=False)
```

- **Functionality:** This function creates and prepares the data loaders for the training, validation, and test datasets.
    - **Dataset Class:** It first creates a custom ChessDataset instance, which handles loading and preprocessing of images and their corresponding FEN labels.
    - **Batching:** It then wraps the dataset into a DataLoader object, which handles batching of images, parallel processing (via num_workers), and efficient data transfer (via pin_memory and prefetch_factor).
- **Impact:**
    - **Shuffling and Batching:** Batching helps in training the model efficiently by allowing the GPU to process multiple samples in parallel. Shuffling ensures the model doesn't learn biases from the order of the data.
    - **Parallel Processing:** By utilizing multiple workers (num_workers), data loading becomes faster, allowing the training loop to progress smoothly without bottlenecks caused by slow data loading.

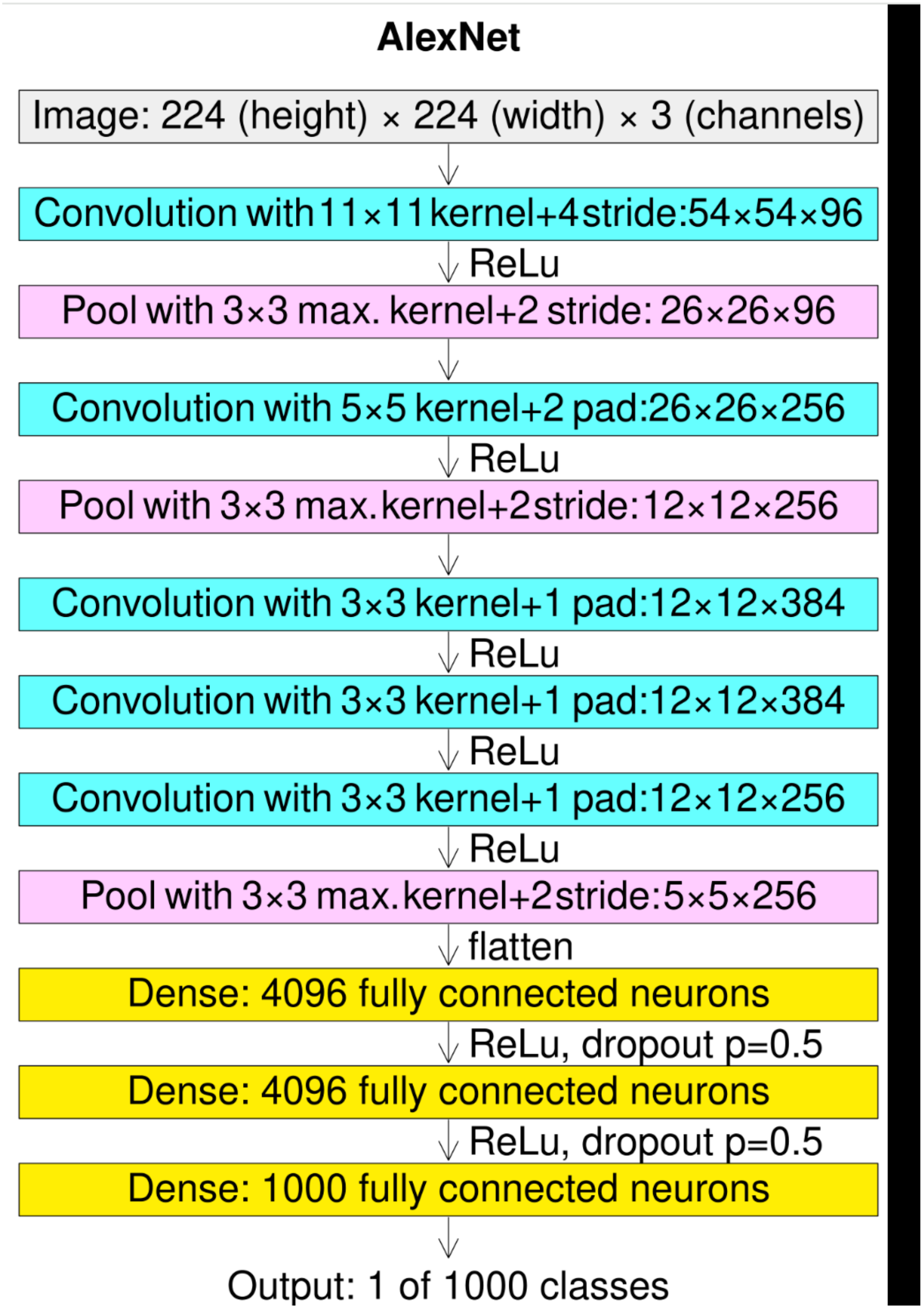
Model Training

```
def get_model(pretrained: bool = True, num_classes: int = cfg.OUTPUTS_A) -> nn.Module:
    model = models.alexnet(pretrained=pretrained)

    for param in model.parameters():
        param.requires_grad = False

    model.avgpool = nn.AdaptiveAvgPool2d((6, 6))
    model.classifier = nn.Sequential(
        nn.Dropout(),
        nn.Linear(256 * 6 * 6, 4096),
        nn.ReLU(inplace=True),
        nn.Dropout(),
        nn.Linear(4096, 4096),
        nn.ReLU(inplace=True),
        nn.Linear(4096, num_classes),
    )

    return model.to(cfg.DEVICE)

print("Initializing EfficientNet-B7 model...")
model = get_model(pretrained=True, num_classes=cfg.OUTPUTS_A)
print("Model initialized successfully.\n")

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()),
lr=cfg.LEARNING_RATE)

scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min',
factor=0.5, patience=2, verbose=True)
```

AlexNet Model:

- **AlexNet**: AlexNet is a convolutional neural network (CNN) that is commonly used for image classification tasks. In my implementation, the model is modified to fit the specific task of classifying chessboard grids into 13 categories (for each piece and an empty grid).
- **Impact**: Powerful pre-trained models are capable of learning complex features from the images, even with relatively limited data.

MODEL ARCHITECTURE


AlexNet
Image: 224 (height) × 224 (width) × 3 (channels)
Convolution with 11×11 kernel+4 stride:54×54×96
ReLu
Pool with 3×3 max. kernel+2 stride: 26×26×96
Convolution with 5×5 kernel+2 pad:26×26×256
ReLu
Pool with 3×3 max.kernel+2stride:12×12×256
Convolution with 3×3 kernel+1 pad:12×12×384
ReLu
Convolution with 3×3 kernel+1 pad:12×12×384
ReLu
Convolution with 3×3 kernel+1 pad:12×12×256
ReLu
Pool with 3×3 max.kernel+2stride:5×5×256
flatten
Dense: 4096 fully connected neurons
ReLu, dropout p=0.5
Dense: 4096 fully connected neurons
ReLu, dropout p=0.5
Dense: 1000 fully connected neurons
Output: 1 of 1000 classes

The original architecture of **AlexNet** is replicated but the last layer is modified to accommodate 13 classes.

Cross-Entropy Loss:

- **Functionality:** Cross-Entropy Loss is used as the loss function for training, which is suitable for multi-class classification tasks like this one, where the model must predict one of 13 classes for each grid.
- **Why it's needed**: Cross-entropy loss measures how well the predicted probabilities match the true labels. This loss function is ideal for classification tasks and helps the model adjust its weights during training to improve accuracy.

Adam Optimizer:

- **Functionality:** Adam (Adaptive Moment Estimation) is used for optimizing the model's parameters. It adapts the learning rate based on the gradients, improving convergence speed.
- **Why it's needed:** Adam is widely used because it combines the advantages of both AdaGrad and RMSProp, making it efficient for training deep learning models. It adjusts the learning rate for each parameter individually, making the training process more stable.

Early Stopping

```
class EarlyStopping:

    def __init__(self, patience=5, verbose=False, delta=0.0, path='checkpoint.pt'):
        self.patience = patience
        self.verbose = verbose
        self.delta = delta
        self.path = path
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss, model):
        if self.best_loss is None:
            self.best_loss = val_loss
            self.save_checkpoint(val_loss, model)
        elif val_loss > self.best_loss - self.delta:
            self.counter += 1
            if self.verbose:
                print(f'EarlyStopping counter: {self.counter} out of {self.patience}')
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_loss = val_loss
            self.save_checkpoint(val_loss, model)
            self.counter = 0

    def save_checkpoint(self, val_loss, model):
        if self.verbose:
            print(f'Validation loss decreased ({self.best_loss:.4f} -->
{val_loss:.4f}).  Saving model ...')
        torch.save(model.state_dict(), self.path)
```

- **Functionality:** Early stopping is used during model training to stop the training process once the validation loss stops improving for a set number of epochs (called patience). This prevents the model from continuing to learn when it is no longer improving, which can lead to overfitting on the training data.

  Impact:

- **Prevents Overfitting:** By stopping training early when the model starts to overfit (i.e., when the validation loss begins to rise after a period of improvement), it ensures that the model generalizes better to unseen data.

Checkpointing

- The EarlyStopping class not only tracks validation loss but also saves the model whenever the validation loss improves (using save_checkpoint).
- The model is saved to the specified path (e.g., best_model.pth) each time the validation loss is better than the previously recorded loss. The saved model state can later be reloaded to resume training or for inference without having to retrain the model from scratch.

plot_metrics Function

```
def plot_metrics(history: dict, save_path: str):
    epochs = range(1, len(history['train_loss']) + 1)

    plt.figure(dpi=300)
    plt.plot(epochs, history['train_loss'], '-o', label='Train Loss')
    plt.plot(epochs, history['val_loss'], '-o', label='Validation Loss')
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Training and Validation Loss')
    plt.savefig(os.path.join(save_path, "train_valid_loss.png"))
    plt.close()
    print(f"Saved loss plot to {os.path.join(save_path, 'train_valid_loss.png')}")

    plt.figure(dpi=300)
    plt.plot(epochs, history['train_accuracy'], '-o', label='Train Accuracy')
    plt.plot(epochs, history['val_accuracy'], '-o', label='Validation Accuracy')
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.title('Training and Validation Accuracy')
    plt.savefig(os.path.join(save_path, "train_valid_accuracy.png"))
    plt.close()
    print(f"Saved accuracy plot to {os.path.join(save_path,
'train_valid_accuracy.png')}")
```

- **Functionality:** This function visualizes and saves the training and validation metrics (loss and accuracy) over epochs**.**
    - **Plotting Loss and Accuracy:** It plots the training and validation loss, and training and validation accuracy for each epoch, helping you track the model's progress.
    - **Saving Figures:** After plotting the metrics, it saves these figures as .png files for further analysis or reporting.

evaluate_model Function

```
def evaluate_model(model: nn.Module, dataloader: DataLoader) -> Tuple[float, float,
float, float]:
    model.eval()
    correct = 0
    total = 0
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for inputs, labels in tqdm(dataloader, desc='Testing', leave=False):
            inputs = inputs.to(cfg.DEVICE, non_blocking=True)
            labels = labels.to(cfg.DEVICE, non_blocking=True)
            with torch.cuda.amp.autocast():
                outputs = model(inputs)
                _, preds = torch.max(outputs, 1)

            correct += torch.sum(preds == labels.data)
            total += labels.size(0)

            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    accuracy = correct.double() / total
    precision, recall, f1, _ = precision_recall_fscore_support(all_labels, all_preds,
average='weighted',
                                                              zero_division=0)

    print(f'Accuracy of the network on the {total} test grids: {accuracy * 100:.2f}
%')
    print(f'Precision: {precision:.4f}, Recall: {recall:.4f}, F1-Score: {f1:.4f}')

    return accuracy.item(), precision, recall, f1
```

- **Functionality: This function evaluates the trained model on the test set.**
  - **Evaluation Mode:** It sets the model to evaluation mode (model.eval()), ensuring layers like dropout or batch normalization work in inference mode.
  - **Predictions:** It processes each image in the test dataset, makes predictions, and compares them to the true labels.
  - **Metrics:** The function calculates the accuracy, precision, recall, and F1-score for the model's predictions.

make_prediction Function

```
def make_prediction(model: nn.Module, device: str, image_path: str, transform=None) ->
str:
    model.eval()
    image = cv2.imread(image_path)
    if image is None:
        print(f"Warning: Image {image_path} could not be read. Using blank grid.")
        grids = [np.zeros((cfg.IMAGE_SIZE, cfg.IMAGE_SIZE, cfg.CHANNELS),
dtype=np.uint8)] * 64
    else:
        image_resized = cv2.resize(image, (400, 400))
        grids = split_image_into_grids(image_resized, grid_size=(50, 50))
        if len(grids) < 64:
            print(f"Warning: Image {image_path} has less than 64 grids. Padding with 
blank grids.")
            grids += [np.zeros((50, 50, cfg.CHANNELS), dtype=np.uint8)] * (64 - 
len(grids))
        elif len(grids) > 64:
            grids = grids[:64]

    predicted_labels = []

    if transform is None:
        transform = A.Compose([
            A.Resize(height=cfg.IMAGE_SIZE, width=cfg.IMAGE_SIZE),  # Ensure resizing
            A.Normalize(mean=(0.485, 0.456, 0.406),
                        std=(0.229, 0.224, 0.225)),
            ToTensorV2()
        ])

    for grid in grids:
        if len(grid.shape) == 2:  # If grayscale, convert to RGB
            grid_rgb = cv2.cvtColor(grid, cv2.COLOR_GRAY2RGB)
        else:
            grid_rgb = cv2.cvtColor(grid, cv2.COLOR_BGR2RGB)
        pil_image = Image.fromarray(grid_rgb)
        transformed = transform(image=np.array(pil_image))
        input_tensor = transformed['image']
        input_tensor = input_tensor.unsqueeze(0).to(device)

        with torch.cuda.amp.autocast():
            with torch.no_grad():
                outputs = model(input_tensor)
                _, pred = torch.max(outputs, 1)
                predicted_labels.append(pred.item())

    fen_notation = label_to_fen(predicted_labels)
    return fen_notation
```

- **Functionality:** This function takes an uploaded chessboard image and uses the trained model to predict the FEN notation.
    - **Image Preprocessing:** It first converts the image into grayscale, splits it into 64 grids, and resizes the grids to the required input size.
    - **Model Prediction:** For each grid, it makes a prediction using the trained model and translates the output (predicted labels) back into FEN notation using the label_to_fen function.

create_excel_file Function

```python
def create_excel_file(image_paths: List[str], labels: List[int], excel_path: str):
    data = []
    unique_images = list(set(image_paths))  # Ensure unique images
    for img_path in unique_images:
        fileName = os.path.splitext(os.path.basename(img_path))[0]
        fen = fileName
        # Extract labels corresponding to this image
        image_labels = [label for path, label in zip(image_paths, labels) if path ==
img_path]
        # Convert list of labels to a comma-separated string
        labels_str = ','.join(map(str, image_labels))
        data.append([img_path, fen, labels_str])

    df = pd.DataFrame(data, columns=['image_path', 'fen', 'labels'])
    df.to_excel(excel_path, index=False)
    print(f"Excel file created at {excel_path}")
```

- **Functionality:** This function generates an Excel file containing the predicted FEN notations for the test set images.
  - **Collecting Data:** It collects the image paths, actual FEN notations (from filenames), and the predicted FEN notations.
  - **Excel File Creation:** The function then saves this information as a structured Excel file (.xlsx).
- **Impact:**
  - **Reporting and Record Keeping:** This allows for easy sharing and analysis of the predictions. It provides a tangible output for the user or team to evaluate the model’s predictions.

Streamlit App Development

```python
import streamlit as st
import cv2
import torch
import numpy as np
from PIL import Image
import albumentations as A
from albumentations.pytorch import ToTensorV2

# Constants
IMAGE_SIZE = 224
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
LABEL_TO_FEN_SHORT = {
    0: "0", 1: "p", 2: "P",
    3: "b", 4: "B",
    5: "r", 6: "R",
    7: "n", 8: "N",
    9: "q", 10: "Q",
    11: "k", 12: "K"
}

# Define your model architecture
def define_model(num_classes=13):
    from torchvision.models import vit_b_16
    model = vit_b_16(weights=None)  # Do not load pretrained weights
    num_ftrs = model.heads.head.in_features
    model.heads.head = torch.nn.Linear(num_ftrs, num_classes)  # Match your training
setup
    return model

# Load the trained model weights
@st.cache_resource
def load_model(model_path):
    model = define_model(num_classes=13)
    model.load_state_dict(torch.load(model_path, map_location=DEVICE))
    model.to(DEVICE)
    model.eval()
    return model

# Splitting the image into grids
def split_image_into_grids(image):
    h, w = image.shape
    grid_h, grid_w = h // 8, w // 8
    grids = []
    for i in range(0, h, grid_h):
        for j in range(0, w, grid_w):
            grid = image[i:i + grid_h, j:j + grid_w]
            grids.append(grid)
    return grids

# Convert labels to FEN notation
def label_to_fen(label_list):
    fen = ""
    empty_count = 0
    for idx, label in enumerate(label_list):
        if label == 0:
            empty_count += 1
        else:
            if empty_count > 0:
                fen += str(empty_count)
                empty_count = 0
            fen += LABEL_TO_FEN_SHORT.get(label, "0")
        if (idx + 1) % 8 == 0:
            if empty_count > 0:
                fen += str(empty_count)
                empty_count = 0
            if idx != len(label_list) - 1:
                fen += "/"
    return fen

# Preprocessing the image
transform = A.Compose([
    A.Resize(height=IMAGE_SIZE, width=IMAGE_SIZE),
    A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
    ToTensorV2(),
])

# Predict FEN for a chessboard image
def predict_fen(model, image):
    image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)  # Convert to grayscale
    grids = split_image_into_grids(image_gray)
    if len(grids) != 64:
        raise ValueError(f"Image does not have 64 grids (got {len(grids)}). Check
cropping.")

    predicted_labels = []
    for grid in grids:
        grid_resized = cv2.resize(grid, (IMAGE_SIZE, IMAGE_SIZE))
        grid_rgb = cv2.cvtColor(grid_resized, cv2.COLOR_GRAY2RGB)
        input_tensor =
transform(image=np.array(grid_rgb))["image"].unsqueeze(0).to(DEVICE)
        with torch.no_grad():
            outputs = model(input_tensor)
            _, pred = torch.max(outputs, 1)
            predicted_labels.append(pred.item())

    return label_to_fen(predicted_labels)

# Streamlit Application
st.title("Chess FEN Prediction App")
st.write("Upload a chessboard image to predict the FEN representation.")

# Upload the model file
model_path =
"/Users/bhoomikan/Documents/Deep_Learning/Project/VIT_early_stop_best_model.pth"
if not model_path:
    st.warning("Please provide the path to your trained model.")
```

```python
# Load the model
if model_path:
    try:
        model = load_model(model_path)
        st.success("Model loaded successfully!")
    except Exception as e:
        st.error(f"Error loading model: {e}")

# Image upload
uploaded_image = st.file_uploader("Upload a chessboard image (JPEG/PNG):",
type=["jpg", "jpeg", "png"])

if uploaded_image is not None:
    # Read and display the uploaded image
    file_bytes = np.asarray(bytearray(uploaded_image.read()), dtype=np.uint8)
    image = cv2.imdecode(file_bytes, cv2.IMREAD_COLOR)
    st.image(cv2.cvtColor(image, cv2.COLOR_BGR2RGB), caption="Uploaded Image",
use_column_width=True)

    # Predict FEN
    if st.button("Predict FEN"):
        try:
            predicted_fen = predict_fen(model, image)
            st.subheader("Predicted FEN Notation:")
            st.text(predicted_fen)
        except Exception as e:
            st.error(f"Error during prediction: {e}")
```

- **Image Upload and Prediction:**
  - **Functionality:** The Streamlit app allows users to upload images of chessboards. The model then processes these images and predicts the corresponding FEN notation by classifying each grid on the board.
  - **Why it's needed:** This interface is crucial for making the model accessible to users who do not have programming skills. It allows users to easily upload chessboard images and obtain predictions without needing to interact with the code directly.

Chess FEN Prediction App

Upload a chessboard image to predict the FEN representation.

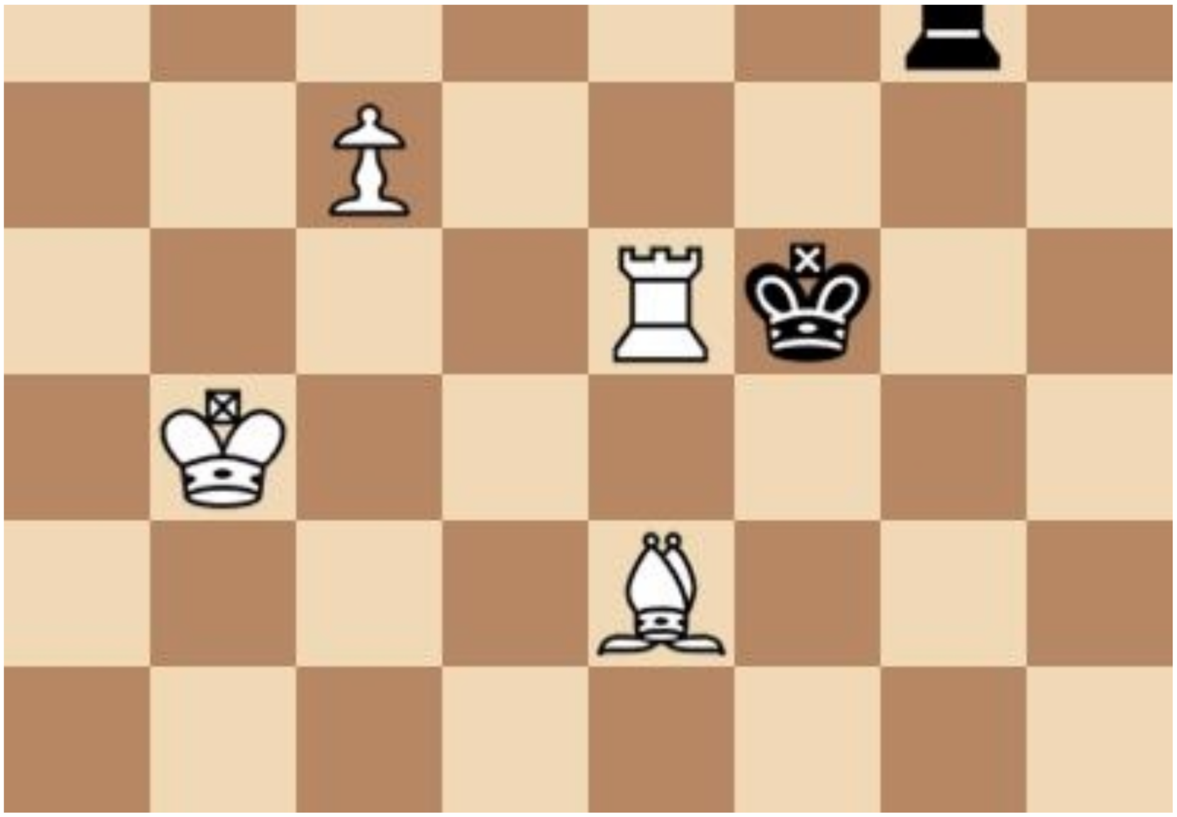
Model loaded successfully!

Upload a chessboard image (JPEG/PNG):

Drag and drop file here

Limit 200MB per file • JPG, JPEG, PNG

Browse files


Uploaded Image

Predict FEN

Predicted FEN Notation:

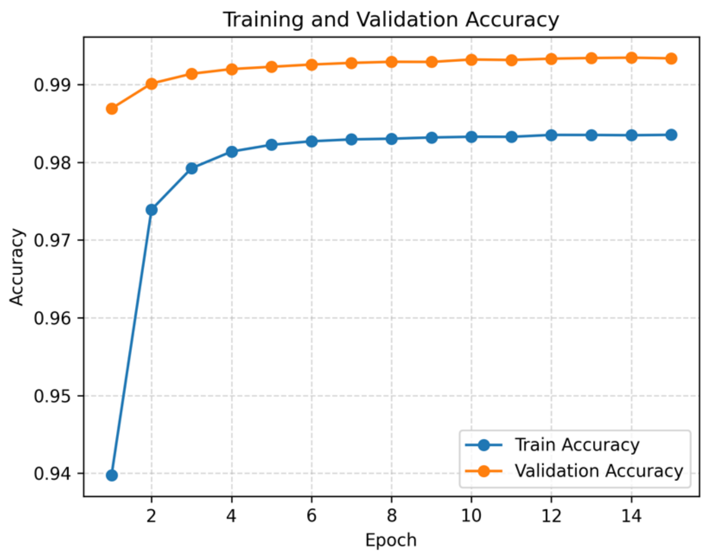
1b1B1Qr1/7p/6r1/2P5/4Rk2/1K6/4B3/8

Instructions

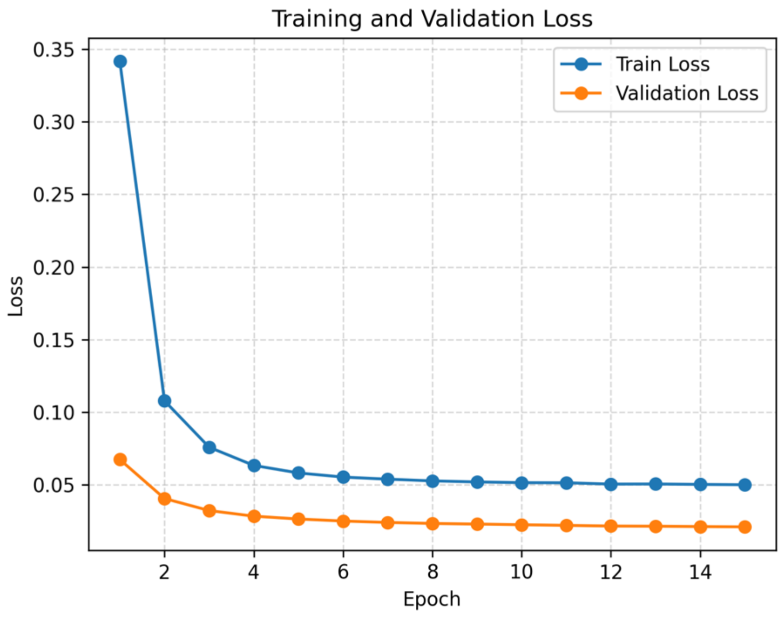
Here replace the model_path with your own model_path.
Link to download the best model to run this streamlit app(https://github.com/AshwinMuthuraman/Deep_Learning_Project_6303_10/blob/sarva/VIT_early_stop_best_model.pth)

I just developed the basic streamlit app to predict FEN using VIT model. This code was optimized by my other project teammates.

Results :

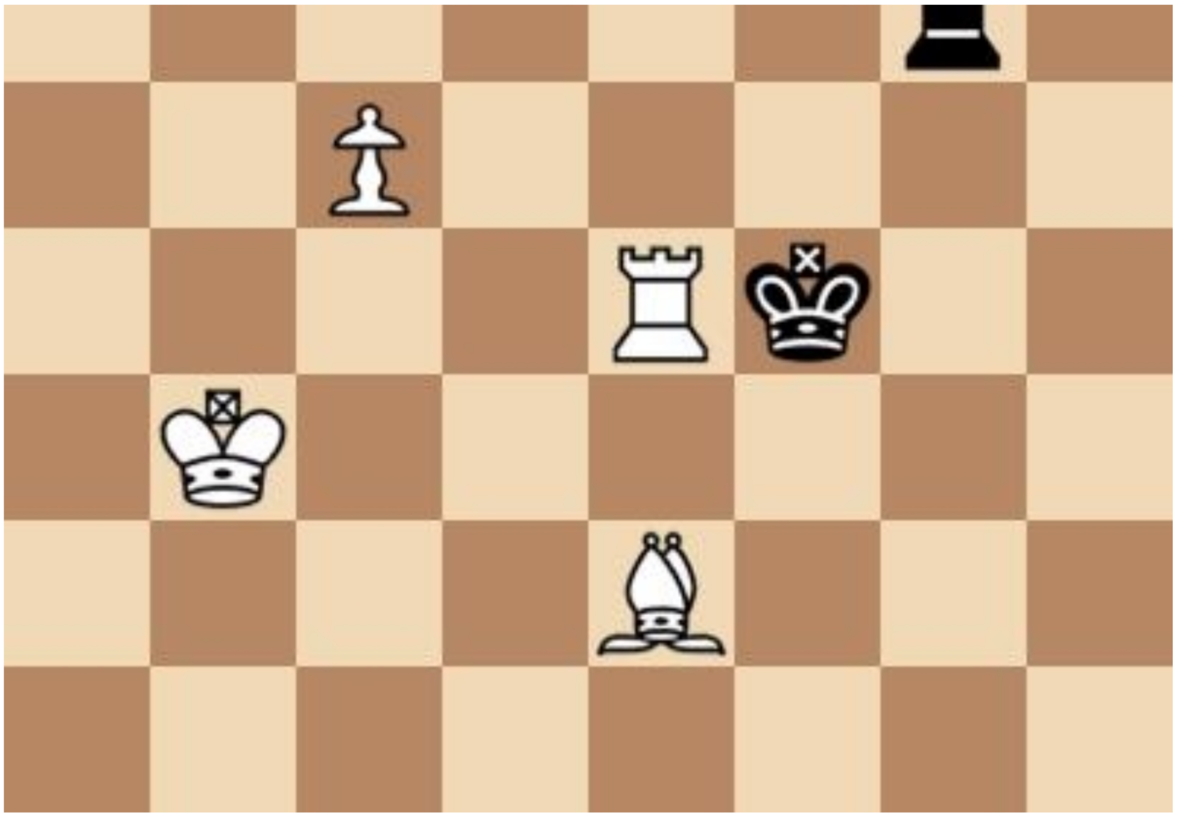


- **Training and Validation Accuracy Plot**
  - The training accuracy improves consistently over epochs, nearing 0.999, showing the model is learning well on the training data.
  - Validation accuracy is slightly higher than training accuracy and stabilizes around 0.9995, indicating the model is generalizing well without overfitting.



- **Training and Validation Loss Plot**
  - The training loss decreases significantly over epochs, stabilizing at a very low value, indicating the model has fit the training data effectively.
  - Validation loss remains low and stable, showing good generalization and no signs of significant overfitting.

StreamLit App Predictions Using VIT Trained Model Weights

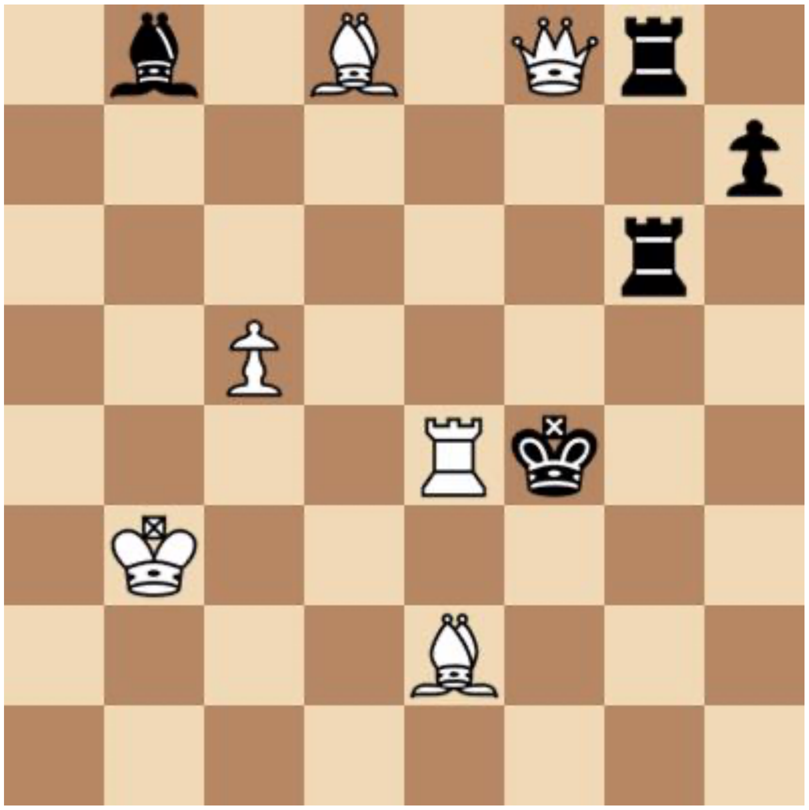


Uploaded Image

Predicted FEN Notation:

1b1B1Qr1/7p/6r1/2P5/4Rk2/1K6/4B3/8

Original Image:



Let's break down the FEN string:

FEN String: 1b1B1Qr1/7p/6r1/2P5/4Rk2/1K6/4B3/8

- **Rank 8** (1b1B1Qr1):
  - 1: One empty square.
  - b: Black bishop.
  - 1: One empty square.
  - B: White bishop.
  - 1: One empty square.
  - Q: White queen.
  - r: Black rook.
  - 1: One empty square.

Rank 7 (7p):

- 7: Seven empty squares.
- p: Black pawn.

Rank 6 (6r1):

- 6: Six empty squares.
- r: Black rook.
- 1: One empty square.

Rank 5 (2P5):

- 2: Two empty squares.
- P: White pawn.
- 5: Five empty squares.

Rank 4 (4Rk2):

- 4: Four empty squares.
- R: White rook.
- k: Black king.
- 2: Two empty squares.

Rank 3 (1K6):

- 1: One empty square.
- K: White king.
- 6: Six empty squares.

Rank 2 (4B3):

- 4: Four empty squares.
- B: White bishop.
- 3: Three empty squares.

Rank 1 (8):

- 8: Eight empty squares (an empty rank).

Summary And Conclusions:

- **Data Augmentation Impact:** Augmentation techniques (such as rotation, flipping, and grid distortion) significantly enhanced the model's ability to generalize, reducing overfitting and improving performance on unseen data.
- **Model Choice:** AlexNet, though a simpler architecture, performed well for this task, but more advanced models like Vision Transformers (ViT) provided even better results due to their ability to capture long-range dependencies.

- **Effectiveness of Early Stopping:** Early stopping was crucial in preventing overfitting by halting training once validation loss stopped improving, thus saving computational resources and enhancing model generalization.
- **Image Preprocessing Importance:** Resizing and normalization were key to making the images compatible with deep learning models and ensuring that the network learned meaningful patterns efficiently.
- **Understanding of FEN Conversion:** Converting predicted grid labels into Forsyth-Edwards Notation (FEN) required efficient label mapping and careful post-processing to maintain board integrity.
- **Evaluation Metrics:** Precision, recall, and F1 score metrics were essential for evaluating model performance, especially for handling the multiclass classification task where imbalanced class distributions could be a challenge.

Improvements:

- **Scalability Challenges:** The Streamlit app was a good prototype but could benefit from optimizations to handle larger datasets and real-time processing more efficiently.
- Integrate more diverse data sources for enhanced training.
- **Model Interpretability:** Integrate model explainability techniques such as Grad-CAM to provide insights into how the model makes predictions, making it more transparent and trustworthy for users.

Code Utilization:

Train File:

Original Lines From Internet: 400

Modified Lines: 60

Added Lines: 200 (train_and_validate, Plot_Metrics, create_excel_file, main)

Therefore, the percentage of the code that copied from the internet = 57%

Streamlit File:

Original Lines From Internet: 80

Modified Lines: 10

Added Lines: 5

Therefore, the percentage of the code that copied from the internet = 82%

References:

https://www.analyticsvidhya.com/blog/2020/08/top-4-pre-trained-models-for-image-classification-with-python-code/

https://medium.com/@salmantahir717/a-comprehensive-examination-of-pre-trained-models-alexnet-vgg-16-vgg-19-resnet-googlenet-and-02036a70c4c1

https://ieeexplore.ieee.org/document/9019943

https://www.researchgate.net/publication/347125306_LiveChess2FEN_a_Framework_for_Classifying_Chess_Pieces_based_on_CNNs