DEEP LEARNING REPORT

Ashwin Muthuraman

G46745441

1. Introduction. An overview of the project and an outline of the shared work.

This project focused on developing a Chess FEN number generator utilizing various pretrained models to ascertain the most effective one for accurate FEN predictions. A key component of the project was the implementation of a user-friendly interface via Streamlit, allowing users to upload an image of a chessboard or capture it live through a webcam. This interface is instrumental in predicting the FEN number based on the uploaded or live-captured image.

To tackle the challenges posed by diverse backgrounds in user-uploaded images, we integrated a YOLO algorithm to enhance object detection capabilities. This was crucial in ensuring the accuracy of the FEN predictions by effectively identifying chess pieces across varying conditions.

My primary contributions encompassed optimizing the EfficientNet B7 model for enhanced training outcomes and refining the YOLO algorithm object detection. Meanwhile, my teammates were responsible for developing the baseline code for Streamlit, preprocessing tasks, and managing other model integrations

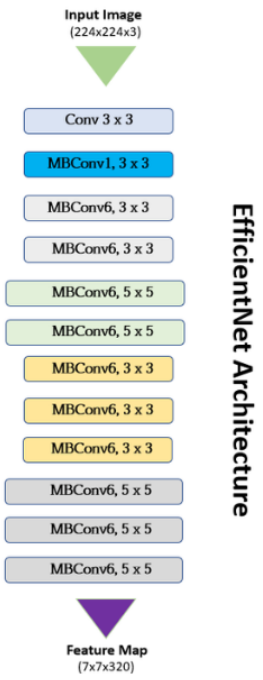
2. Description of your individual work.

EfficientnetB7

Architecture:

EfficientNet's architecture is based on a mobile inverted bottleneck convolution (MBConv) block, which consists of:

1. **Depthwise Convolution:** Applies a convolution to each input channel independently, reducing computational cost.
2. **Expansion Layer:** Increases the number of channels.
3. **Depthwise Convolution**: Another depthwise convolution.
4. **Projection Layer:** Reduces the number of channels back to the original input channels.



The reason I chose EfficientNet is because of its compound scaling method that uniformly scales network depth, width, and resolution. This is achieved using the following formula:

width × depth^2 × resolution^2 ≈ constant

Here,

- width refers to the number of channels in a layer.
- depth is the number of layers.
- resolution is the input image resolution.

By keeping this product constant, EfficientNet balances the scaling of these dimensions to achieve optimal performance and efficiency.

Code Modifications:

In the code I developed, the EfficientNet-B7 model was used as a pre-trained backbone. The final classification layer was replaced with a custom linear layer with 13 output units, corresponding to the 13 different chess pieces and the empty grid. This modification allows the model to classify chess pieces accurately based on the input images.

YOLOv7 for object detection

I used existing yolov7 code to train the model for the chess dataset. Since an image can have excess background while uploading or while showing it via the webcam. I developed the following functionalities to overcome it.

- **load_yolo_model:** This function loads the pre-trained YOLOv7 model from a specified file path.
- **non_max_suppression:** This function is a common technique in object detection to filter out redundant detections. It takes a list of detected bounding boxes with their confidence scores and class probabilities. It then iteratively selects the box with the highest confidence score and suppresses any overlapping boxes with higher Intersection over Union (IoU) than a specified threshold. This helps to reduce false positives and improve the overall accuracy of the detection.
- **xywh2xyxy:** This function converts the bounding box format from (center x, center y, width, height) to (top-left x, top-left y, bottom-right x, bottom-right y), which is a more common format for visualizing and processing bounding boxes.
- **rescale_coords:** This function rescales the bounding box coordinates from the model's output space to the original image space, taking into account the image resizing and padding that might have been applied during preprocessing.
- **run_yolo_inference:** This function takes an image as input, preprocesses it, feeds it to the YOLOv7 model, and returns the detected objects with their bounding boxes and confidence scores.
- **letterbox:** Resizes and pads an image to a fixed size.
- **draw_yolo_boxes:** Draws bounding boxes and labels on an image
- **process_chessboard:** Extracts and warps a chessboard region from an image.
- **VideoProcessor**: A class that processes video frames, performs YOLOv7 detection, and processes detected chessboards.

3. **Why Use These Functions?**

These functions are essential for implementing object detection with YOLOv7:

- **load_yolo_model:** To load the pre-trained model and make it ready for inference.
- **non_max_suppression**: To improve the quality of detections by filtering out redundant and low-confidence predictions.

- **xywh2xyxy:** To convert the bounding box format to a more convenient format for visualization and further processing.
- **rescale_coords:** To ensure that the detected bounding boxes accurately correspond to the original image, especially after resizing and padding.
- **run_yolo_inference:** To perform the entire object detection pipeline, from preprocessing the image to generating the final detections.
- **letterbox:** Needed to ensure consistent input size for YOLOv7, improving model performance and accuracy.
- **draw_yolo_boxes:** Visualizes the model's output, making it easier to understand and debug.
- **process_chessboard:** Prepares the chessboard image for further analysis, such as FEN (Forsyth-Edwards Notation) generation.
- **VideoProcessor:** Integrates the functions above into a real-time video processing pipeline, allowing for continuous object detection and chessboard analysis.

3. Detailed Explanation of individual contribution.

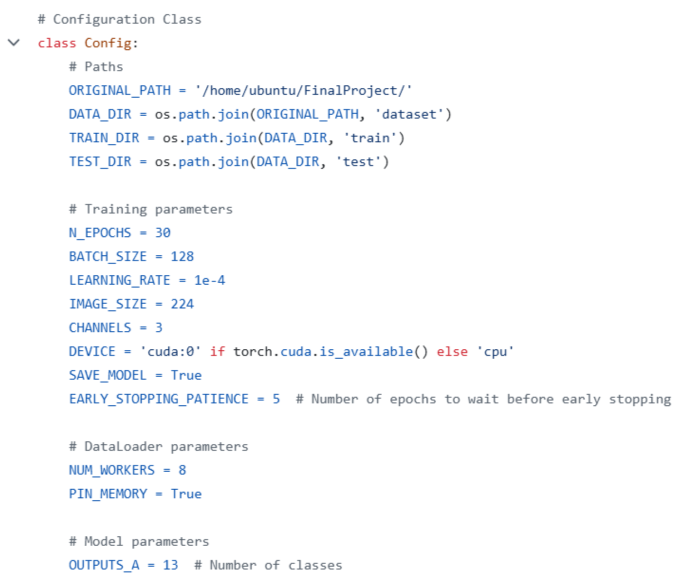
EfficientNet B7 Training

My primary contribution to the project involved the detailed setup and training of the EfficientNet B7 model, with a particular focus on hyperparameter tuning and performance optimization.

Hyperparameter Tuning: To achieve optimal results, I adjusted various hyperparameters. Key settings included:

- **Hyperparameters:**
  - Batch Size: 128
  - Learning Rate: 0.0001
  - Epochs: 30
  - Image Size: 224x224 pixels
- **Activation Function:**
  - Uses Swish activation function as default in EfficientNet-B7.
- **Loss Function:**
  - Cross-Entropy Loss for multi-class classification.
- **Early Stopping:**

   - Stops training if validation loss does not improve after 5 epochs (patience = 5).
- **Learning Rate Scheduler:**
   - ReduceLROnPlateau: Reduces learning rate when validation loss plateaus.
- **Model Customization:**
   - Replaces EfficientNet-B7 classifier with a custom layer for 13 classes (chess pieces and empty grid).
- **Model Saving and Metrics Plotting:**
   - Saves the best model based on validation loss.
   - Plots training/validation loss and accuracy metrics.



```
# Configuration Class
class Config:
    # Paths
    ORIGINAL_PATH = '/home/ubuntu/FinalProject/'
    DATA_DIR = os.path.join(ORIGINAL_PATH, 'dataset')
    TRAIN_DIR = os.path.join(DATA_DIR, 'train')
    TEST_DIR = os.path.join(DATA_DIR, 'test')

    # Training parameters
    N_EPOCHS = 30
    BATCH_SIZE = 128
    LEARNING_RATE = 1e-4
    IMAGE_SIZE = 224
    CHANNELS = 3
    DEVICE = 'cuda:0' if torch.cuda.is_available() else 'cpu'
    SAVE_MODEL = True
    EARLY_STOPPING_PATIENCE = 5  # Number of epochs to wait before early stopping

    # DataLoader parameters
    NUM_WORKERS = 8
    PIN_MEMORY = True

    # Model parameters
    OUTPUTS_A = 13  # Number of classes
```

Model Training and Validation Loop: The training loop included detailed logging of both training and validation metrics, which were critical for monitoring the model's performance and making necessary adjustments. The implementation of early stopping based on validation loss was a key technique to ensure efficient training cycles:

```python
# ==============================
# Training and Validation Function
# ==============================

def train_and_validate(
        model: nn.Module,
        train_loader: DataLoader,
        val_loader: DataLoader,
        criterion: nn.Module,
        optimizer: torch.optim.Optimizer,
        scheduler: torch.optim.lr_scheduler._LRScheduler,
        num_epochs: int = cfg.N_EPOCHS,
        patience: int = cfg.EARLY_STOPPING_PATIENCE,
        save_path: str = cfg.MODEL_SAVE_NAME
) -> dict:
    """
    Train and validate the model.
    """
    # Initialize early stopping
    early_stopping = EarlyStopping(patience=patience, verbose=True, path=save_path)

    best_val_loss = float('inf')

    # Lists to store metrics
    history = {
        'train_loss': [],
        'train_accuracy': [],
        'val_loss': [],
        'val_accuracy': []
    }

    for epoch in range(num_epochs):
        start_time = time.time()

        # Training Phase
        model.train()
        train_loss = 0.0
        correct_train = 0
        total_train = 0

        for images, labels in tqdm(train_loader, desc=f"Epoch {epoch + 1}/{num_epochs} - Training"):
            images = images.to(cfg.DEVICE, non_blocking=True)
            labels = labels.to(cfg.DEVICE, non_blocking=True)

            optimizer.zero_grad()

            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Backward pass and optimization
            loss.backward()
            optimizer.step()

            # Statistics
            train_loss += loss.item() * images.size(0)
            _, preds = torch.max(outputs, 1)
            correct_train += torch.sum(preds == labels.data)
            total_train += labels.size(0)

        avg_train_loss = train_loss / len(train_loader.dataset)
        train_accuracy = correct_train.double() / len(train_loader.dataset)

        # Validation Phase
        model.eval()
        val_loss = 0.0
        correct_val = 0
        total_val = 0

        with torch.no_grad():
            for images, labels in tqdm(val_loader, desc=f"Epoch {epoch + 1}/{num_epochs} - Validation"):
                images = images.to(cfg.DEVICE, non_blocking=True)
                labels = labels.to(cfg.DEVICE, non_blocking=True)

                outputs = model(images)
                loss = criterion(outputs, labels)

                val_loss += loss.item() * images.size(0)
                _, preds = torch.max(outputs, 1)
                correct_val += torch.sum(preds == labels.data)
                total_val += labels.size(0)

        avg_val_loss = val_loss / len(val_loader.dataset)
        val_accuracy = correct_val.double() / len(val_loader.dataset)

        # Scheduler step
        scheduler.step(avg_val_loss)

        # Record history
        history['train_loss'].append(avg_train_loss)
        history['train_accuracy'].append(train_accuracy.item())
        history['val_loss'].append(avg_val_loss)
        history['val_accuracy'].append(val_accuracy.item())

        # Print metrics
        epoch_time = (time.time() - start_time) / 60  # in minutes
        print(
            f"Epoch [{epoch + 1}/{num_epochs}] "
            f"| Train Loss: {avg_train_loss:.4f} "
            f"| Train Acc: {train_accuracy:.4f} "
            f"| Val Loss: {avg_val_loss:.4f} "
            f"| Val Acc: {val_accuracy:.4f} "
            f"| Time: {epoch_time:.2f} min"
        )

        # Early Stopping
        early_stopping(avg_val_loss, model)
        if early_stopping.early_stop:
            print("Early stopping triggered.")
            break

        # Save the best model
        if avg_val_loss < best_val_loss:
            best_val_loss = avg_val_loss
            if cfg.SAVE_MODEL:
                torch.save(model.state_dict(), save_path)
                print(f"Best model saved with val_loss: {best_val_loss:.4f}")

    print('Training complete')
    print(f'Best Validation Loss: {best_val_loss:.4f}')

    # Save training and validation metrics plots
    os.makedirs(cfg.PLOTS_SAVE_PATH, exist_ok=True)
    plot_metrics(history, cfg.PLOTS_SAVE_PATH)
    model.load_state_dict(torch.load(save_path, map_location=cfg.DEVICE))
    return history
```

YOLO Object Detection Techniques

For the YOLO (You Only Look Once) component of the project, I undertook the following steps:

1. **Data Acquisition:**
    - Sourced a comprehensive dataset of chess boards and chess pieces from Roboflow Universe.
    - The dataset, titled "2D Chessboard and Chess Pieces," was downloaded, including both images and corresponding labels. https://universe.roboflow.com/chess-project/2d-chessboard-and-chess-pieces
2. **Model Training:**
    - Utilized the YOLOv7 architecture for object detection.
    - Executed the training process using the following command:

```
python3 train.py --workers 8 --device 0 --batch-size 32 --data data/data.yaml --img 640 640 --cfg cfg/training/yolov7.yaml --weights '' --name yolov7 --hyp data/hyp.scratch.p5.yaml
```

3. **Code Source:**

- The training script (train.py) was obtained from the official YOLOv7 GitHub repository. https://github.com/WongKinYiu/yolov7/blob/main/train.py

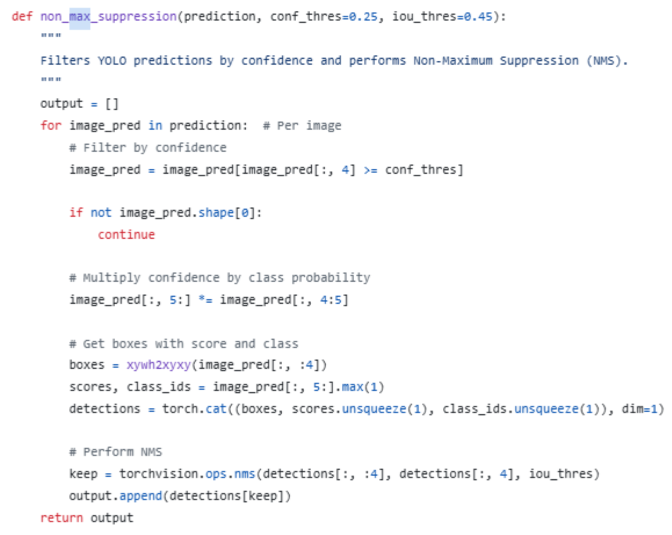
Real-Time Detection Setup in Streamlit: To manage live chessboard detection, I established a real-time video processing pipeline using WebRTC within Streamlit. This setup involved configuring the video stream to apply the YOLO model on-the-fly to incoming video frames, allowing for dynamic object detection:

```python
class VideoProcessor(VideoProcessorBase):
    def __init__(self, yolo_model, fen_model):
        self.yolo_model = yolo_model
        self.fen_model = fen_model
        self.fen_results = []  # Initialize as an empty list

    def recv(self, frame):
        img = frame.to_ndarray(format="bgr24")
        original_img = img.copy()

        # YOLOv7 Inference
        detections = run_yolo_inference(self.yolo_model, img, conf_thres=YOLO_CONF_THRESHOLD,
                                        iou_thres=YOLO_IOU_THRESHOLD, device=DEVICE)

        # Draw Bounding Boxes
        annotated_image = draw_yolo_boxes(img.copy(), detections, YOLO_CLASS_NAMES)

        # Process Each Detection
        for det_idx, det in enumerate(detections, 1):
            cls = int(det[5])
            if YOLO_CLASS_NAMES[cls] != "chessboard":
                continue  # Skip if not a chessboard

            # Apply the corner detection and perspective transform pipeline
            processed_img, warped = process_chessboard(img, det)
            if warped is not None:
                # Predict FEN
                fen = predict_fen_from_image(self.fen_model, warped)
                self.fen_results.append(fen)

                # Overlay FEN annotation near the bounding box on the annotated image
                x1, y1, x2, y2 = map(int, det[:4])
                # Position the text slightly above the top-left corner of the bounding box
                text_position = (x1, y1 - 10 if y1 - 10 > 10 else y1 + 10)
                cv2.putText(annotated_image, fen, text_position,
                            cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 255), 2)

                # Limit the number of stored FEN results to the last 10
                if len(self.fen_results) > 10:
                    self.fen_results.pop(0)

        # Convert back to VideoFrame
        return av.VideoFrame.from_ndarray(annotated_image, format="bgr24")
```

Image Preprocessing for YOLO: Standardizing input images is crucial for consistent detection. I developed functions to resize and pad images to fit YOLO's input requirements, ensuring that every image, regardless of its original size, is processed correctly:

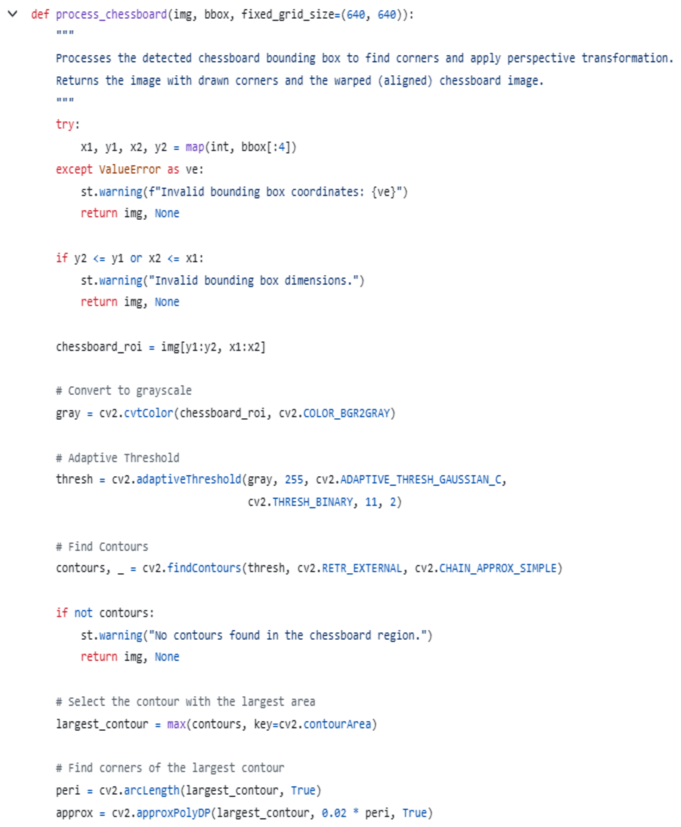
```python
def letterbox(img, new_shape=(640, 640), color=(114, 114, 114)):
    """
    Resizes and pads an image to fit the desired shape.
    """
    shape = img.shape[:2]  # Current shape [height, width]
    ratio = min(new_shape[0] / shape[0], new_shape[1] / shape[1])  # Ratio  = new / old
    new_unpad = int(round(shape[1] * ratio)), int(round(shape[0] * ratio))  # W, H
    dw, dh = new_shape[1] - new_unpad[0], new_shape[0] - new_unpad[1]  # Padding
    dw, dh = dw // 2, dh // 2

    img = cv2.resize(img, new_unpad, interpolation=cv2.INTER_LINEAR)
    img = cv2.copyMakeBorder(img, dh, dh, dw, dw, cv2.BORDER_CONSTANT, value=color)
    return img, ratio, (dw, dh)
```

Detection Post-Processing: After YOLO identifies potential chessboard areas, I applied Non-Maximum Suppression (NMS) to refine these detections, selecting the most probable bounding box while eliminating overlaps and redundancies:



```python
def non_max_suppression(prediction, conf_thres=0.25, iou_thres=0.45):
    """
    Filters YOLO predictions by confidence and performs Non-Maximum Suppression (NMS).
    """
    output = []
    for image_pred in prediction:  # Per image
        # Filter by confidence
        image_pred = image_pred[image_pred[:, 4] >= conf_thres]

        if not image_pred.shape[0]:
            continue

        # Multiply confidence by class probability
        image_pred[:, 5:] *= image_pred[:, 4:5]

        # Get boxes with score and class
        boxes = xywh2xyxy(image_pred[:, :4])
        scores, class_ids = image_pred[:, 5:].max(1)
        detections = torch.cat((boxes, scores.unsqueeze(1), class_ids.unsqueeze(1)), dim=1)

        # Perform NMS
        keep = torchvision.ops.nms(detections[:, :4], detections[:, 4], iou_thres)
        output.append(detections[keep])
    return output
```

Chessboard Contour Detection and Perspective Transformation: Post-detection, the identified chessboard regions undergo further processing using OpenCV's contour detection to accurately locate the board's edges. This is followed by a perspective transformation to normalize the board's orientation for consistent analysis:



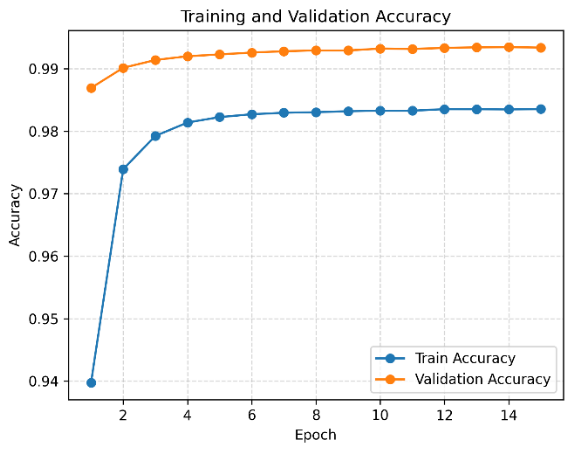
```python
def process_chessboard(img, bbox, fixed_grid_size=(640, 640)):
    """
    Processes the detected chessboard bounding box to find corners and apply perspective transformation.
    Returns the image with drawn corners and the warped (aligned) chessboard image.
    """
    try:
        x1, y1, x2, y2 = map(int, bbox[:4])
    except ValueError as ve:
        st.warning(f"Invalid bounding box coordinates: {ve}")
        return img, None

    if y2 <= y1 or x2 <= x1:
        st.warning("Invalid bounding box dimensions.")
        return img, None

    chessboard_roi = img[y1:y2, x1:x2]

    # Convert to grayscale
    gray = cv2.cvtColor(chessboard_roi, cv2.COLOR_BGR2GRAY)

    # Adaptive Threshold
    thresh = cv2.adaptiveThreshold(gray, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                   cv2.THRESH_BINARY, 11, 2)

    # Find Contours
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    if not contours:
        st.warning("No contours found in the chessboard region.")
        return img, None

    # Select the contour with the largest area
    largest_contour = max(contours, key=cv2.contourArea)

    # Find corners of the largest contour
    peri = cv2.arcLength(largest_contour, True)
    approx = cv2.approxPolyDP(largest_contour, 0.02 * peri, True)
```

4. Results

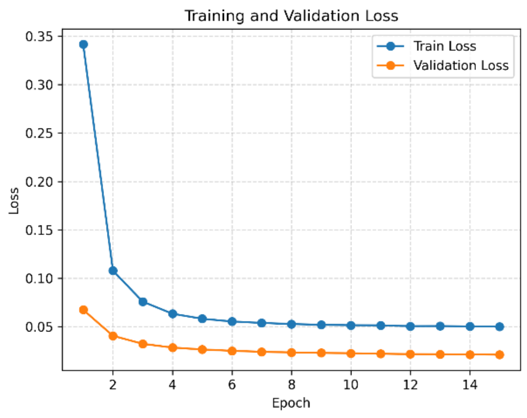
A. Training and Validation Performance

The experiments conducted as part of the project demonstrated impressive model performance, as illustrated in the attached training and validation accuracy and loss plots for the EfficientNet B7 model.

Training and Validation Accuracy Plot: This plot illustrates a steady increase in training accuracy, reaching a plateau at approximately 98% accuracy. The validation accuracy closely tracks the training accuracy, maintaining a consistent level near 99%, which indicates that the model generalizes well on unseen data.



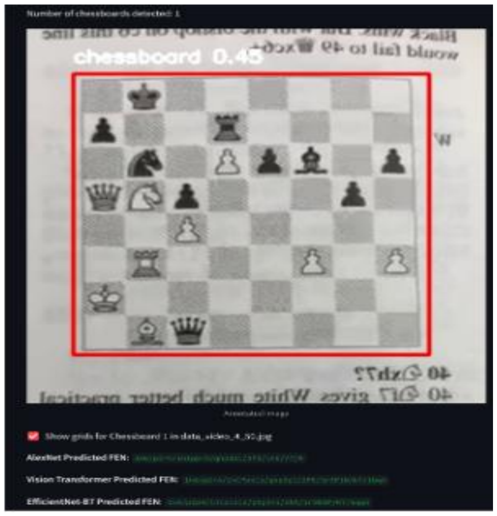
Training and Validation Loss Plot: The loss plot reveals a sharp decline in training loss, stabilizing at around 0.05, while the validation loss remains low, underlining the model's effective learning and generalization capabilities.



B. YOLO Object Detection and FEN Prediction

YOLO Detection Output: The YOLO model successfully detects the chessboard within the image, as shown by the red bounding box. This initial detection is crucial for accurate FEN prediction, as it isolates the board from any background noise or interference.

FEN Prediction: The system's ability to predict FEN notations from these detections demonstrates a seamless integration of object detection with pattern recognition, crucial for applications like digital chess analysis where accurate real-time predictions are valuable.



Number of chessboards detected: 1

Annotated image

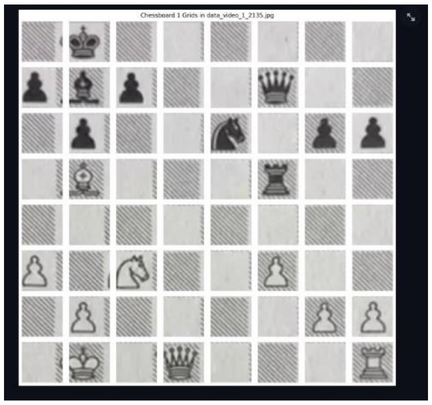
Show grids for Chessboard 1 in data_video_4_50.jpg

AlexNet Predicted FEN: [illegible]

Vision Transformer Predicted FEN: [illegible]

EfficientNet-B7 Predicted FEN: [illegible]

Chessboard Grid Detection: Following the detection of the chessboard, the model further processes the image to identify individual squares. Each square is delineated with a white bounding box, as illustrated in the second image. This grid-based breakdown is essential for analyzing the presence and type of chess pieces on each square to generate the FEN notation.



5. Summary and conclusions:

The project achieved significant results in developing a Chess FEN number generator using deep learning techniques. Here's a summary of the key outcomes and insights:

1. **Model Performance:**
    - The EfficientNet B7 model demonstrated excellent performance, reaching a training accuracy of 98% and a validation accuracy of 99%.
    - The loss plots showed a sharp decline in training loss, stabilizing at around 0.05, with consistently low validation loss.
2. **Object Detection:**
    - The YOLO model successfully detected chessboards within images, isolating them from background noise.
    - Accurate chessboard grid detection was achieved, with individual squares delineated for piece analysis.
3. **FEN Prediction:**
    - The system demonstrated the ability to predict FEN notations from detected chessboards, showcasing successful integration of object detection and pattern recognition

Key Learnings:

1. **Hyperparameter Tuning:** The importance of careful hyperparameter adjustment was evident in optimizing model performance.
2. **Real-time Processing:** Implementing a real-time video processing pipeline using WebRTC within Streamlit proved effective for live chessboard detection.
3. **Image Preprocessing:** Standardizing input images was crucial for consistent detection across various image sources.

Future Improvements:

1. **Enhanced Dataset**: Expanding the training dataset with more diverse chessboard images could improve model robustness.
2. **Advanced Augmentation**: Implementing more sophisticated data augmentation techniques might further enhance model generalization.
3. **Model Ensemble:** Exploring ensemble methods combining multiple models could potentially increase prediction accuracy.

4. **Performance Optimization:** Further optimization of the real-time processing pipeline could improve speed and efficiency for live detection.

6. Percentage of the code

EfficientnetB7 training

- **Total lines from internet:** 160 (model definition and training and validation function)
- **Lines modified:** 70 (changed the model definition and train and val function according to our use case)
- **Lines added:** 40 (implemented Early stopping class)

Lines of code used directly: 160 - 70 = 90

Total lines of code in the final version: 90 (used directly) + 70 (modified) + 40 (added) = 200

Percentage of code from internet: (90 / 200) * 100 = 45%

Yolov7 with streamlit

- Total lines of code from internet: 270 (model definition, train and validation function)
- Lines modified: 110 (model definition, train and validation function according to our use case)
- Lines added: 60 (early stopping class)

Total lines of code: 270 + 110 + 60 = 440

Percentage of code from internet: (270 / 440) * 100 ≈ 61.36%

7. References

https://github.com/WongKinYiu/yolov7/blob/main/train.py (for yolov7 training)

https://docs.streamlit.io/ (for developing the stream lit application)