Overview of the shared work:

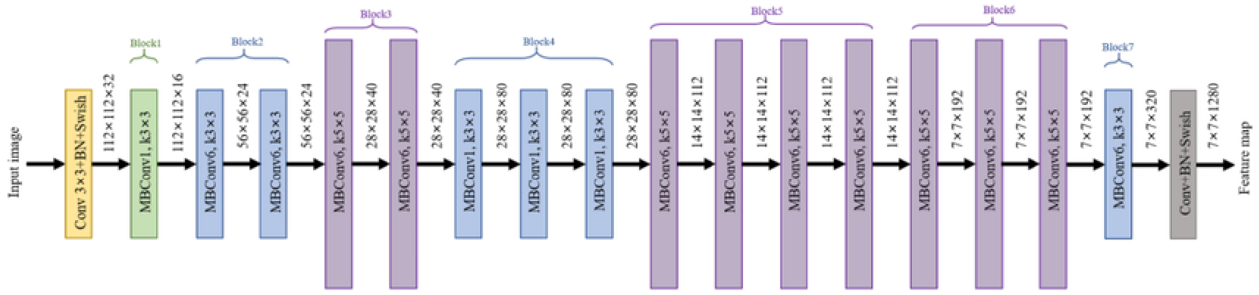
We built a model to recognize chess positions from a schematic image of a chess board. Our model can generate FEN descriptions (Forsyth-Edwards Notation) based on the position of the pieces on the board. Here's a breakdown of what we did:

- **Dataset:**
  - We created a dataset of 100,000 images containing randomly generated chess positions of 5-15 pieces (2 kings and 3-13 pawns/pieces).
  - The images were generated using 28 styles of chess boards and 32 styles of chess pieces.
  - We split the dataset into 80,000 training images and 20,000 testing images.
- **Model:**
  - We used a pre-trained EfficientNet-B0, EfficientNet-B7, Vision Transformer and ALexnet model with adjusted hyperparameters for image classification.
  - Our model was trained to identify a 64x64 grid within the image, and then classify each grid to predict the chess piece on that square.
- **Evaluation:**
  - We evaluated our model's performance on a separate test set and reported accuracy, precision, recall, and F1 score.
- **Integration:**
  - We then built a U-Net and Yolo model to get just the chess-board from an image
  - The final product was integrated into Streamlit for an interactive user interface.

Individual work:

Part 1: Classification modeling

I developed a model using the EfficientNet-B0 architecture.



EfficientNet-B0 is composed of several building blocks, primarily utilizing **mobile inverted bottleneck convolution (MBConv)** layers with depthwise separable convolutions. It also incorporates squeeze-and-excitation (SE) blocks to enhance channel-wise feature interactions.

The architecture can be divided into seven blocks:

1. **Stem Layer (Block 1)**:
    - **Input:** The network starts with an input image (e.g., 224×224×3).
    - **Convolutional Layer:** A standard 3 × 3 convolution with a stride of 2 is applied to reduce the spatial resolution.
    - **Activation Function:** Swish activation function is used to introduce non-linearity.
    - **Output:** Feature maps of size 12 × 112 × 32.

    $y = x \cdot \sigma(x)$, where $\sigma(x) = 1 / (1 + e^{-x})$ (Swish activation)

2. **MBConv Blocks (Block 2 to Block 6)**:
    - These blocks are the core of the architecture. Each MBConv block applies an inverted residual bottleneck structure consisting of:
        - **Expansion Phase:** Expands the input channels by a factor (e.g., 6) to increase feature richness.
        - **Depthwise Convolution:** Applies depthwise separable convolutions to reduce computational cost while preserving spatial dimensions.
        - **Squeeze-and-Excitation (SE):** Weights the channels adaptively.
        - **Projection Phase:** Projects the expanded features back to the original channel dimensions.
    - Multiple MBConv blocks with varying kernel sizes (3 × 3 or 5 × 5) and strides are stacked to extract hierarchical features.
    - Output feature sizes progressively reduce while the number of channels increases across blocks.
3. MBConv mathematical representation:

    y = Projection(Depthwise(Expansion(x)))

4. **Final Layers (Block 7)**:
    - A 7×7 convolution layer aggregates spatial information globally, producing feature maps of size 7 × 7 × 320.
    - Batch normalization is applied, followed by Swish activation.
5. **Classification Head**:
    - Global average pooling compresses the feature maps into a single vector per channel.
    - A fully connected layer (dense) maps the feature vector to class probabilities.

    $\hat{y} = \text{Softmax}(W \cdot \text{GAP}(x) + b)$

Modeling Process

Initially, I initialized the EfficientNet-B0 model, leveraging pretrained weights from ImageNet to capitalize on previously learned features. The model's final classifier layer was customized to output predictions for our specific number of classes. I focused the training process by freezing the base convolutional layers of the network and allowing only the classifier layers to update during training. This approach helped in fine-tuning the model to our dataset while leveraging the robust feature-extraction capabilities that EfficientNet-B0 offers.

Hyperparameters

The model training was configured with the following hyperparameters:

- **Epochs**: I set the model to train for 15 epochs.
- **Batch Size**: I used a batch size of 32 to balance between computational efficiency and model performance.
- **Learning Rate**: The initial learning rate was set at 0.001, using an Adam optimizer for adaptive learning rate adjustments.
- **Early Stopping**: Implemented with a patience of 5 epochs to prevent overfitting by halting the training if the validation loss did not improve.
- **Learning Rate Scheduler**: Employed a ReduceLROnPlateau scheduler, reducing the learning rate by 50% when the validation loss plateaued, which helped in refining the model's training towards the later stages.

Model Performance

The model achieved outstanding results on the test dataset, underlining the effectiveness of the chosen architecture and training strategy:

- **Test Accuracy**: Achieved an impressive accuracy of 99.93%.
- **Class-wise Performance**: The precision, recall, and F1-scores across classes were highly satisfactory, with most classes exhibiting nearly perfect scores. Notably, class-wise metrics were as follows:
    - Class 0 (example class): Precision=1.0000, Recall=1.0000, F1-Score=1.0000
    - Other classes demonstrated similarly high performance metrics, indicating the model's robust ability to generalize across different types of image data.
- **Overall F1 Scores**:
    - **F1 Micro**: 0.9993, reflecting the average score per instance across classes.
    - **F1 Macro**: 0.9961, illustrating the average score per class, confirming the balanced nature of the model across diverse categories.

Part 2: U-Net

After successfully generating FEN numbers from perfectly cropped images, we considered real-world scenarios where images might be imperfectly cropped or have backgrounds. To address this, we implemented two approaches: U-Net for image segmentation and Yolo for object detection. These techniques allow us to extract perfectly cropped chessboards from imperfect input images, ensuring accurate classification.

But the problem was we did not have such a dataset. So I decided to prepare such a dataset using augmentation:

Augmentation Goals My objective was to enhance the dataset for training a U-Net segmentation model by creating three augmented datasets:

- **Augmented Training Set**: Utilized the original training images as masks and generated various transformations for augmentation.
- **Augmented Validation Set**: Derived from a subset of the training data with unique augmentations for validation purposes.
- **Augmented Testing Set**: Augmented the testing images separately, using the original images as masks.

Augmentation Details I employed a range of transformations to add variability:

- **Background Padding**: Introduced random solid color backgrounds and real-world indoor scenes from Places365.
- **Rotation**: Applied random rotations between ±15 degrees, carefully filling in the rotated areas to prevent black artifacts.
- **Brightness, Contrast, and Hue Adjustments**: Made subtle changes to enhance realism.
- **Random Text and Numbers**: Added randomly around the chessboard without overlapping.
- **Scaling and Positioning**: Resized and repositioned chessboards on the backgrounds.
- **Final Resolution**: Standardized all images to 400x400 pixels.

Augmentation Pipeline I processed each dataset distinctly:

- **Training Dataset**: Started with 25,000 original images, generating six augmentations per image.
- **Validation Dataset**: Used 10,000 original images, creating three augmentations per image.
- **Testing Dataset**: Employed all 20,000 testing images, producing two augmentations per image.

And each dataset will be augmented differently to create variability:

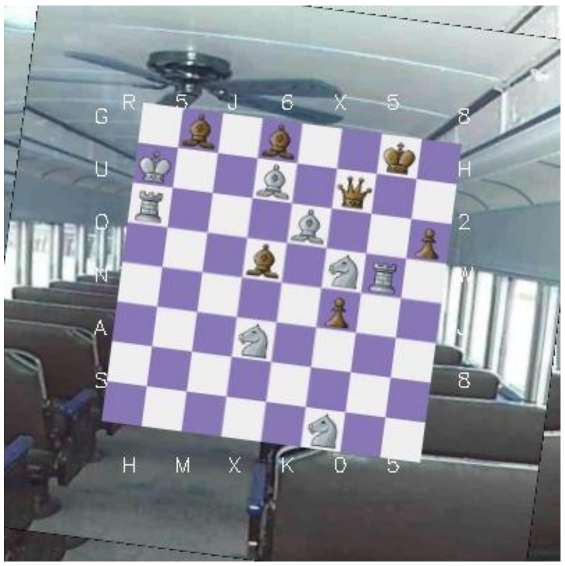
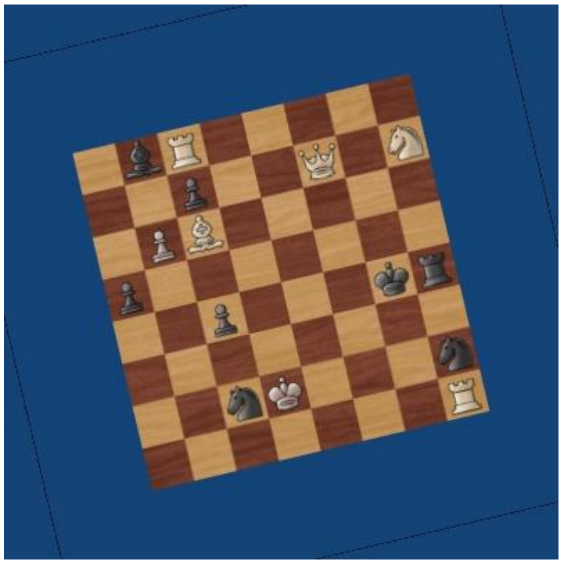
1. **Augmented Training Images**:
    - Use **random solid color padding** or **Places365 backgrounds**.
    - Rotate the chessboard around a random center.
    - Add **random single digits (0-9)** or **letters (A-Z)** around the chessboard on all four sides.
2. **Augmented Validation Images**:
    - Same as the training images, but with **fewer augmentation variations** for consistency:
        - Use random **solid color padding only**.
        - Apply a **fixed range of rotation** (e.g., ±15°).
        - Add random **digits or letters**, but with **minimal randomness in font size and color**.
3. **Augmented Testing Images**:
    - Different augmentations to simulate **real-world testing scenarios**:
        - Use **indoor images from Places365** exclusively for padding.
        - Randomly rotate the chessboard within a wide range (e.g., ±45°).
        - Add **no numbers or letters** around the chessboard to test raw performance.

Verification I ensured each augmented image had a corresponding mask and maintained consistent naming conventions. Regular sanity checks validated dataset sizes and augmentation quality.

Reduced Dataset Creation: Due to the lengthy training times, I opted to create a reduced training dataset:

- **Augmented Training Set**: Contained 65,004 images.
- **Masks**: Included 10,834 masks to match all augmented images.

Here are samples of how the augmented images looked like:

Model Training: I prepared and trained a U-Net model.

U-Net Architecture

U-Net is a convolutional neural network primarily designed for biomedical image segmentation tasks, though it has been widely adopted in other fields requiring precise localization and segmentation. It follows an encoder-decoder structure, with skip connections that help preserve spatial information.

Overview of the Architecture

The U-Net architecture can be divided into two main parts: the **contracting path (encoder)** and the **expanding path (decoder)**. Together, they form a "U"-shaped structure. Skip connections between the encoder and decoder paths allow the network to combine low-level spatial information with high-level semantic features.

1. Contracting Path (Encoder)

The contracting path is responsible for extracting features from the input image. It consists of repeated applications of:

- **Two 3×3 convolutions**: Each convolution is followed by a ReLU activation function and batch normalization.
- **Max Pooling 2×2**: Reduces the spatial dimensions by half while retaining the most significant features.
- **Feature Map Expansion**: The number of feature maps doubles at each step to capture more complex representations.

At the end of this path, the spatial resolution is significantly reduced, but feature richness is maximized.

Mathematically:

$f_{(conv)} = \text{ReLU}(\text{Conv}(f_{input}) + b)$

2. Bottleneck

The bottleneck sits between the encoder and decoder, where the smallest resolution but the highest number of feature maps exist. It acts as a bridge between the two paths.

It involves:

- Two 3×3 convolutions.
- ReLU activations.
- Dropout or other regularization techniques (optional, depending on the implementation).

---

3. Expanding Path (Decoder)

The expanding path reconstructs the segmentation map from the encoded feature maps. It consists of repeated applications of:

- **Transposed Convolutions (Up-convolutions 2×2)**: Increases spatial dimensions by upsampling.

- **Concatenation**: Skip connections from the corresponding encoder layer are concatenated with the upsampled feature maps to combine high-resolution spatial information with semantic context.
- **Two 3×3 Convolutions**: Refines the features after upsampling.

The output dimensions gradually increase, matching the input resolution.

4. Skip Connections

Skip connections directly connect the encoder layers to the corresponding decoder layers. These connections:

- Help preserve spatial information lost during downsampling.
- Facilitate better gradient flow during backpropagation.
- Combine low-level features (from the encoder) with high-level features (from the decoder).

Mathematically:

$f_{(decoder)} = \text{Concat}(f_{(encoder)}, f_{(upconv)})$

5. Output Layer

The final output layer is a 1×1 convolution that maps the feature maps to the desired number of output classes (e.g., binary segmentation for two classes). The softmax or sigmoid activation function is applied to produce probabilities for each class.

$p = \text{Softmax}(\text{Conv}_{1\times1}(f\_decoder))$

1. Model overview

The U-Net architecture consists of an **encoder-decoder** structure:

- **Encoder (contracting path):** Extracts high-level features while progressively reducing the spatial dimensions of the input image.
- **Bottleneck:** Acts as a bridge between the encoder and decoder, learning compact feature representations.
- **Decoder (expanding path):** Restores spatial dimensions and refines feature maps to reconstruct the segmented output.
- **Skip Connections:** Links corresponding layers in the encoder and decoder to retain fine-grained spatial details to ensure the retention of fine-grained details.

2. Encoder (Downsampling Path)

The encoder comprises a sequence of convolutional blocks, each followed by a downsampling operation using **max pooling**. Each convolutional block consists of:

- Two 3×3 convolutional layers.
- **ReLU activation** for non-linear feature extraction.
- **Output Channels:**
    - Block 1: Input (3 channels) → Output (64 channels)
    - Block 2: Input (64 channels) → Output (128 channels)
    - Block 3: Input (128 channels) → Output (256 channels)
    - Block 4: Input (256 channels) → Output (512 channels)

3. Bottleneck

The bottleneck acts as the transition between the encoder and decoder:

- Contains two 3×3 convolutional layers with 1024 output channels.
- Applies ReLU activation after each convolution.

4. Decoder (Upsampling Path)

The decoder reconstructs the segmentation map by progressively upsampling feature maps and concatenating them with the corresponding encoder outputs via **skip connections**:

- Each decoder block performs the following steps:
    - **Upsampling:** Uses a transposed convolution layer to double the spatial dimensions.
    - **Concatenation:** Combines the upsampled feature map with the feature map from the corresponding encoder block.
    - **Convolutional Processing:** Passes the concatenated output through two 3×3 convolutional layers with ReLU activation.
- **Output Channels:**
    - Block 1: Input (1024 + 512 channels) → Output (512 channels)
    - Block 2: Input (512 + 256 channels) → Output (256 channels)
    - Block 3: Input (256 + 128 channels) → Output (128 channels)
    - Block 4: Input (128 + 64 channels) → Output (64 channels)

5. Final Output Layer

- A **1×1 convolutional layer** maps the 64-channel output of the decoder to a single-channel segmentation map.
- A **sigmoid activation function** is applied to generate probabilities for each pixel, indicating the likelihood of belonging to the target class.

6. Key Features

- **Skip Connections:** Preserve spatial details lost during downsampling by concatenating encoder feature maps with upsampled decoder feature maps.
- **Upsampling with Transposed Convolutions:** Doubles the spatial resolution at each decoder stage, ensuring a gradual reconstruction of the segmentation map.
- **Symmetry:** The encoder and decoder are symmetric in structure, making U-Net effective for precise localization.

7. Loss Function

- **Binary Cross-Entropy Loss (BCELoss):** Measures the difference between predicted probabilities and the ground truth, suitable for binary segmentation tasks.

8. Advantages

- **High Spatial Precision:** Skip connections ensure the retention of fine-grained details, crucial for tasks requiring precise segmentation.
- **Efficient Training:** The encoder-decoder structure, combined with skip connections, leads to better gradient flow and faster convergence.
- **Scalability:** The architecture can handle various image sizes by adjusting the number of convolutional layers or feature map dimensions.

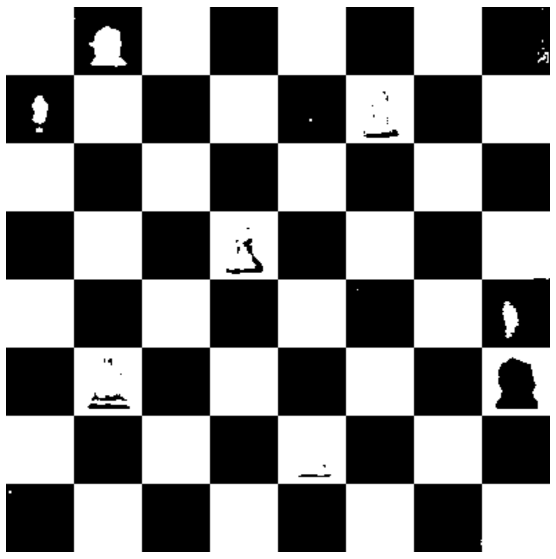
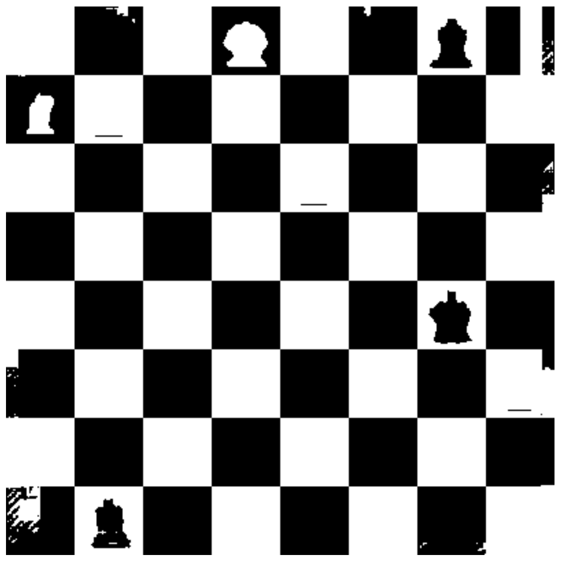
Challenges Faced

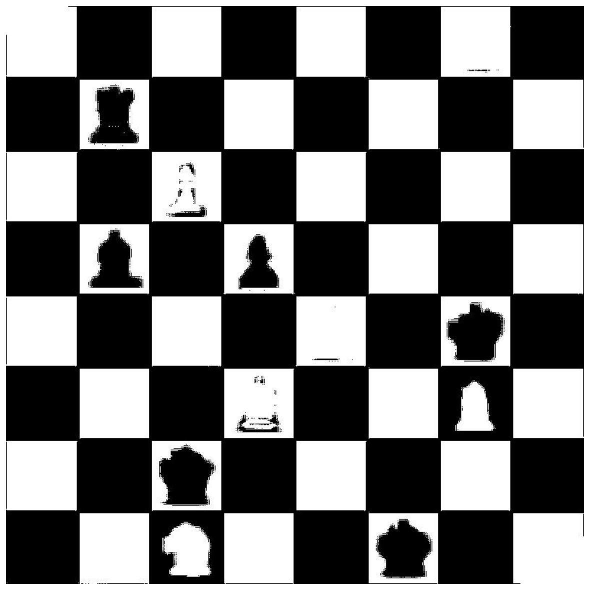
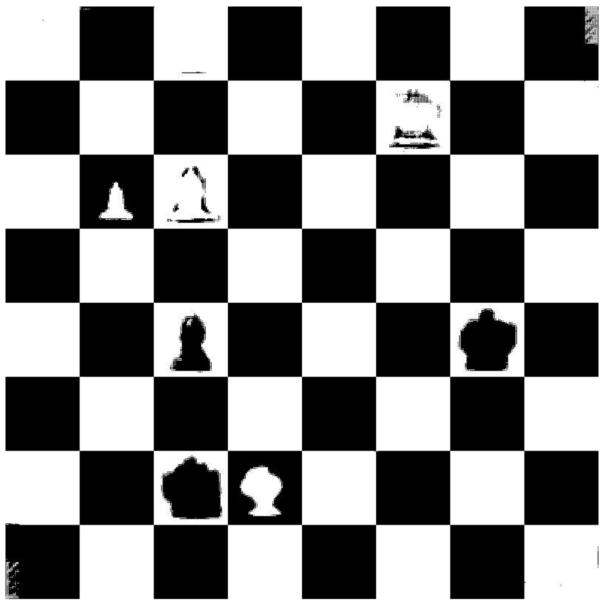
- **Augmentation Issues**: My initial scripts occasionally mismatched images with their corresponding masks. I resolved this by tweaking the augmentation pipelines to ensure proper alignment and consistency.
- **GPU Memory Constraints**: To accommodate the limits of my GPU memory, I was compelled to reduce the batch sizes. This adjustment helped in managing the computational resources more effectively during training.
- **Model Convergence Issues**: The model struggled with slow convergence and suboptimal validation loss performance. The validation loss started at 0.56 and showed minimal improvement, only decreasing to 0.51 after 12 epochs.

Although the U-Net model achieved a satisfactory Dice coefficient of 0.78, which is generally considered satisfactory for segmentation tasks, a closer analysis of the outputs reveals significant issues with the segmentation of the chess pieces. The chessboards themselves were segmented correctly in most images, with clear and accurate delineation of the board's gridlines and cells. However, the segmentation of the chess pieces suffered from several shortcomings, as evident in the example outputs provided:

1. **Imperfect Chess Piece Shapes**:
   - The segmented shapes of the chess pieces in both images fail to accurately match the true shapes of the pieces.
   - For instance:
     - In the first image, some pieces are represented as irregular blobs or are fragmented, which misrepresents their actual contours.
     - Similarly, in the second image, the segmentation of certain pieces shows incomplete or distorted shapes.
2. **Missed Details**:
   - Finer details of the chess pieces, such as their bases or crowns, are often missing in the segmentation results.
   - This lack of detail compromises the model's ability to distinguish between different types of chess pieces.
3. **Over-Smoothing or Fragmentation**:
   - In some cases, the segmented pieces appear over-smoothed, losing the sharp edges that define their boundaries.
   - In other cases, pieces are fragmented into multiple disconnected regions, leading to a misinterpretation of the scene.
4. **False Positives and Negatives**:
   - Some areas on the chessboard are incorrectly segmented as pieces (false positives), while some pieces are partially or completely missed (false negatives).

Here are some of the examples of the chess board detected from augmented set by the Unet model:

Hence we decided not to include this in the streamlit application as the classification model cannot give us accurate FEN code without having a proper shape for each chess piece.