1 Transfer Learning Approach

Transfer learning was employed in the implementation to leverage the robust feature extraction capabilities of pre-trained models on ImageNet. This approach aligns with best practices in fine-grained classification tasks, where lower-level features learned from diverse datasets can be effectively repurposed for specialized domains with limited data. The pre-training on ImageNet's 1.2 million images across 1,000 classes provides the model with a strong foundation for recognizing a wide range of visual patterns, which can then be fine-tuned for our specific classification task despite class imbalance challenges Krizhevsky et al. (2012).

Several pre-trained architectures were evaluated for this task, with VGG-16 Simonyan and Zisserman (2015) demonstrating superior performance in our specific classification context. The effectiveness of transfer learning was evident in the rapid convergence and high accuracy achieved even with our relatively limited dataset, demonstrating the potential of this approach for specialized classification tasks with significant class imbalance.

1.1 Common Implementation Strategy

All models except for the custom CNN utilized transfer learning to leverage knowledge from pre-trained networks. The transfer learning strategy included:

- Using models pre-trained on ImageNet as feature extractors
- Fine-tuning the entire network with a reduced learning rate (typically 0.0001 to 0.001)
- Replacing the final classification layer to output binary predictions (2 classes)
- Implementing dropout layers before final classification to prevent overfitting

This approach follows the established pattern that features learned in early layers of convolutional networks are more general and transferable, while later layers become more task-specific.

1.2 Data Preparation and Augmentation

Data augmentation was crucial to address the limited dataset size and class imbalance issues. Following best practices from Cubuk et al., multiple augmentation techniques were applied consistently across all models:

- **Spatial transformations:** Random horizontal flips, rotations (typically 15 degrees), and random/center crops were applied to increase geometric diversity.
- **Color space transformations:** Color jitter with brightness, contrast, and saturation adjustments of 0.2 magnitude was applied to make models robust to illumination variations.
- **Image enhancement:** In some implementations, sharpening filters were applied to improve feature clarity.
- **Normalization:** All images were normalized to match pre-trained model expectations Shin et al..

The augmentation strategy was deliberately more aggressive for the training set compared to validation and test sets, where only resizing, optional cropping, and normalization were applied to maintain evaluation consistency.

These techniques enhance model robustness to natural variations in image appearance, reducing overfitting and improving generalization capability here.

1.3 Image Preprocessing

All images were preprocessed through a standardized pipeline:

Images were resized to match the architecture's expected input dimensions (224×224 pixels for most models, 299×299 pixels for Inception v3). Pixel values were normalized using ImageNet mean values [0.485, 0.456, 0.406] and standard deviations [0.229, 0.224, 0.225], ensuring input distributions aligned with those seen during pre-training here.

1.4 Training Optimization Strategy

To optimize training with limited data, several techniques were employed consistently:

- **Optimizer:** AdamW optimizer with learning rates between 0.0001-0.001 and weight decay of 0.001-0.0005 was used across implementations to provide adaptive learning with regularization here.
- **Learning rate scheduling:** Adaptive learning rate scheduling using either ReduceLROnPlateau or CosineAnnealingLR was implemented across models, reducing learning rates when validation metrics plateaued.
- **Early stopping:** Training was halted when validation accuracy stopped improving for a specified number of epochs (patience = 3-5) to prevent overfitting. Early Stopping - But When?
- **Gradient clipping:** Applied in some implementations to prevent gradient explosions and stabilize training. Due to the small and imbalanced dataset, gradient clipping was implemented to prevent limited images from causing large weight updates. Why gradient clipping accelerates training: A theoretical justification for adaptivity. International Conference on Learning Representations (ICLR) here
- **Loss function:** Cross-entropy loss was used consistently as the optimization objective for the binary classification task.
- **Mixed precision training:** For computationally intensive models like Inception V3, mixed precision training with torch.amp was used to improve computational efficiency.

The combination of these techniques enabled effective learning despite the challenges of limited data and class imbalance, with our best model achieving significantly better performance than traditional machine learning approaches on the same dataset.

1.5 Regularization Techniques

Multiple regularization strategies were employed to handle the limited data size and class imbalance:

- **Dropout:** Layers with rates between 0.3-0.4 were consistently added before final classification layers to reduce overfitting due to our small dataset size Srivastava et al..
- **Weight decay:** L2 regularization with weight decay values between 1e-4 and 1e-3 was applied across all models to prevent overfitting Krogh & Hertz.
- **Batch normalization:** Used in custom CNN implementations to stabilize learning and improve convergence Ioffe and Szegedy.
- **Data splitting:** Train/validation split of 80%/20% was consistently used to provide reliable validation metrics while maximizing training data.
- **Random seeds:** Fixed random seeds (42) were set for PyTorch, NumPy, and Python's random module to ensure reproducibility. Controlling randomness is essential for reliable hyperparameter tuning, performance assessment, and research reproducibility here.

1.6 Addressing Class Imbalance

Our dataset exhibited significant class imbalance, which can degrade model performance by biasing predictions toward the majority class here. To mitigate this challenge, multiple complementary strategies were implemented on the best performing models that included VGG16, and ViT:

- **Class-Weighted Loss Function**
  - Implemented inverse frequency weighting (Cui et al., 2019) [link]
  - Class weights calculation: $\text{class\_weights}[i] = \frac{\text{total\_samples}}{\text{num\_classes} \times \text{label\_counts}[i]}$
    PyTorch implementation: `CrossEntropyLoss` with class weights tensor
- **Weighted Random Sampling**
  - Balanced mini-batches using PyTorch's `WeightedRandomSampler`
  - Sample weights: $\text{samples\_weights} = \text{class\_weights}[\text{label}]$
  - Oversamples minority class and undersamples majority class [link]
  - Uses replacement sampling for effective batch balancing
- **Class-Specific Data Augmentation**
  - Aggressive minority class augmentation (Shorten & Khoshgoftaar, 2019) [link]
  - Minority class transformations include:
    * 30° random rotations
    * Strong color jitter (brightness/contrast/saturation=0.3)
    * Random resized crops (scale=0.7-1.0)
    * Horizontal flips

    Standard augmentation for majority class (15° rotations, milder parameters)

1.7 Dataset Management

To address the challenges of limited data availability, an 80:20 train-validation split was implemented using random split stratification to maintain class distribution across partitions. This approach ensured that the validation set remained representative of the overall dataset while maximizing the samples available for training Kohavi, 1995.

The batch size was set to 16, striking a balance between computational efficiency and optimization stability. Smaller batch sizes can increase gradient noise, which has been shown to act as an implicit regularizer that can improve generalization, particularly beneficial when working with limited training data Keskar et al., 2016, Masters & Luschi, 2018.

1.8 Evaluation Strategy

Model performance was systematically evaluated using:

- **Validation accuracy:** Used during training to select optimal model checkpoints and trigger early stopping or learning rate adjustments.
- **Test accuracy:** Final evaluation metric on the unseen test set to measure generalization performance.
- **Visualization:** Training loss and validation accuracy curves were plotted to analyze model convergence and potential overfitting.
- **Checkpointing:** Best-performing models based on validation accuracy were saved for later evaluation and deployment.

1.9 Model Checkpointing and Evaluation

Our implementation includes a robust evaluation framework with model checkpointing based on validation accuracy. This ensures that we preserve the best-performing model configuration throughout the training process. The model is trained for 20 epochs with early stopping implicitly implemented through best model saving. Performance is evaluated using accuracy on both validation and test sets, providing a comprehensive assessment of model generalization.

2 Model Architectures and Specific Implementations

2.1 VGG-16 Architecture

2.1.1 Theoretical Foundation

VGG-16 is a convolutional neural network architecture developed by Simonyan and Zisserman (2014) at the Visual Geometry Group (VGG) at Oxford, consisting of 16 weight layers including 13 convolutional layers followed by 3 fully connected layers. The architecture is characterized by its simplicity and depth, using small 3×3 convolutional filters stacked in increasing depth, followed by

max pooling layers. With approximately 138 million parameters, VGG-16 provides a strong foundation for feature extraction in computer vision tasks.

The primary advantage of employing VGG-16 for transfer learning in fine-grained classification tasks is its hierarchical feature representation capability, which enables the capture of both low-level features (edges, textures) and high-level semantic features. Pre-trained on the ImageNet dataset containing over 1.2 million images across 1,000 classes, VGG-16 offers robust initialization weights that facilitate effective knowledge transfer to domain-specific tasks with limited training data.

VGG-16 has demonstrated superior performance in fine-grained classification tasks compared to conventional techniques. Recent studies show that VGG-16 with logistic regression achieved 97.14% accuracy on specialized datasets like Leaf12, significantly outperforming traditional approaches that combined color channel statistics, texture features, and classic classifiers which only reached 82.38% accuracy here. For our specific task of gull species classification, the hierarchical feature representation capabilities of VGG-16 proved particularly effective at capturing the subtle differences in wing patterns and morphological features that distinguish between the target species.

2.1.2 Model Adaptation for Fine-Grained Classification

For our specific fine-grained binary classification task with limited data and class imbalance, the VGG-16 architecture was adapted through a targeted modification strategy:

- The pre-trained VGG-16 model was loaded with ImageNet weights.
- The feature extraction layers (convolutional base) were preserved to maintain the rich hierarchical representations learned from ImageNet.
- The original 1000-class classifier was replaced with a custom binary classification head consisting of:
  - A dropout layer with a rate of 0.4 to reduce overfitting.
  - A fully-connected layer mapping from the original 4096 features to 2 output classes.

(**?**) demonstrated that VGG-16 achieves 94.3% accuracy on CUB-200-2011 by fine-tuning only the final three layers, a strategy mirrored in my VGG implementation where the classifier head was replaced while preserving ImageNet-initialized convolutional weights. This approach aligns with successful methodologies in avian species classification using VGG-16 as demonstrated by Brown et al. (2018), where fine-tuning the architecture by modifying the final classification layer enabled the model to retain general feature recognition capabilities while adapting to species-specific visual characteristics here.

2.2 Vision Transformer (ViT) Architecture

2.2.1 ViT for Fine-Grained Classification

Vision Transformers (ViT) have emerged as powerful alternatives to convolutional neural networks for visual recognition tasks. First introduced by Dosovitskiy et al. (Dosovitskiy et al., 2021), ViTs process images as sequences of fixed-size patches, applying transformer-based self-attention mechanisms to model global relationships between image regions. This architecture enables the capture

of long-range dependencies within images, making it particularly suitable for fine-grained classification tasks where subtle distinctions between similar classes may depend on relationships between distant image features.

2.2.2 Vision Transformer Implementation

For our primary approach, a Vision Transformer using transfer learning from a pre-trained model was implemented:

- Base architecture: 'vit_base_patch16_224' pre-trained on ImageNet from the TIMM library (Wightman, 2021)
- Input resolution: 224×224 pixels with 16×16 pixel patches
- Feature dimension: 768-dimensional embeddings
- Adaptation strategy: Replacement of the classification head with a binary classifier while preserving the pre-trained transformer blocks

The model architecture preserves the core self-attention mechanism of ViT while adapting the final classification layer for our specific binary classification task. This approach follows established transfer learning principles for vision transformers (Touvron et al., 2021), leveraging representations learned from large-scale datasets to overcome our limited training data constraints.

2.2.3 Alternative ViT Implementations

In addition to our primary implementation, we explored two attention-enhanced architectures:

InterpretableViT We developed an InterpretableViT model that incorporates explicit attention mechanisms for improved focus on discriminative features:

- Separates the class token from patch tokens
- Applies a learned attention layer to generate importance weights for each patch
- Combines the class token with attention-weighted patch representations
- Employs a multi-layer classifier with dropout regularization

A key advantage of this architecture is its compatibility with gradient-based visualization techniques. By separating the class token from patch tokens and implementing an explicit attention mechanism, the model facilitates more effective application of Grad-CAM (Selvaraju et al., 2017), allowing for visualization of discriminative image regions contributing to classification decisions.

EnhancedViT We also implemented an EnhancedViT that applies attention-based weighting across all tokens:

- Processes all tokens (including class token) through an attention mechanism
- Generates a single attention-weighted feature representation
- Utilizes a specialized classification head with dropout for regularization

This implementation draws from research on token aggregation strategies in vision transformers (Wang et al., 2021), which shows that attention-weighted token aggregation can improve performance in data-limited regimes.

2.3 Inception v3 Architecture

2.3.1 Theoretical Background

Inception v3, developed by Szegedy et al. (2016), represents a sophisticated CNN architecture designed to efficiently capture multi-scale features through parallel convolution pathways with varied kernel sizes. The key innovation in Inception architectures is the utilization of *Inception modules* that process the same input tensor through multiple convolutional paths with different receptive fields, and then concatenate the results. This enables the network to capture both fine-grained local patterns and broader contextual information simultaneously (Szegedy et al., 2016).

Inception v3 builds upon earlier versions with several important architectural improvements:

- Factorized convolutions to reduce computational complexity
- Spatial factorization into asymmetric convolutions (e.g., $1 \times n$ followed by $n \times 1$)
- Auxiliary classifiers that inject additional gradient signals during training
- Batch normalization for improved training stability and faster convergence
- Label smoothing regularization to prevent overconfidence

These design elements collectively enable Inception v3 to achieve high accuracy while maintaining computational efficiency. As demonstrated by Huang et al. (2019), Inception architectures are particularly effective for tasks requiring multi-scale feature extraction, such as discriminating between visually similar biological specimens (Huang et al., 2019).

2.3.2 Model-Specific Implementation Details

Our implementation adapted the pre-trained Inception v3 model for fine-grained gull species classification with the following specific elements:

- **Input Resolution:** Resize operations were performed to 299×299 pixels (the standard input size for Inception v3). The larger input resolution (299×299 vs 224×224 used by VGG16) provides the Inception architecture with more detailed information, potentially beneficial for capturing the subtle wing pattern differences between gull species (Xie et al., 2020).

- **Auxiliary Outputs:** A distinctive aspect of our Inception v3 implementation was the utilization of auxiliary outputs during training. Inception v3's auxiliary classifier, which branches off from an intermediate layer, provides an additional gradient path during backpropagation. This helps combat the vanishing gradient problem and provides regularization with auxiliary loss weight of 0.3 (He et al., 2019).
- **Mixed-Precision Training:** We employed PyTorch's Automatic Mixed Precision (AMP) to accelerate computation while maintaining numerical stability (Micikevicius et al., 2018). This technique allows the use of float16 precision where appropriate, which reduces memory usage and increases computational speed, especially beneficial when training on GPU-constrained environments like Google Colab.

2.4 Residual Network (ResNet-50) Implementation

Residual Networks (ResNet) have revolutionized deep learning architectures by introducing identity shortcut connections that bypass one or more layers, enabling the training of substantially deeper networks He et al., 2016. These skip connections address the degradation problem by allowing gradients to flow more effectively during backpropagation, mitigating the vanishing gradient issue prevalent in very deep neural networks.

For our fine-grained gull species classification task, a transfer learning approach based on the ResNet-50 architecture was implemented. This implementation was motivated by ResNet's demonstrated success in capturing hierarchical features at multiple levels of abstraction, which is particularly valuable for distinguishing the subtle morphological differences between visually similar gull species He et al., 2016b, Zhao et al., 2019.

2.4.1 Architecture-Specific Enhancements

A distinctive aspect of our ResNet-50 implementation was the incorporation of image sharpening as a preprocessing technique. We applied a 3×3 Laplacian sharpening kernel to enhance edge definition and accentuate the subtle diagnostic features crucial for distinguishing between gull species Gonzalez & Woods, 2018. This approach was inspired by research showing that edge enhancement can improve the detection of fine-grained morphological features in avian classification tasks Berg et al., 2014.

The sharpening kernel was systematically applied to both training and evaluation pipelines using OpenCV's filter2D function, ensuring consistent feature enhancement across all dataset partitions. This preprocessing step proved particularly valuable for highlighting distinctive wingtip patterns and subtle plumage characteristics that serve as key discriminative features between the target species Dutta & Zisserman, 2019.

2.5 Custom CNN with Squeeze-and-Excitation Blocks

2.5.1 Architectural Design

To address the challenges of limited data and class imbalance in fine-grained classification, we developed a lightweight custom CNN architecture incorporating attention mechanisms. Our approach employs Squeeze-and-Excitation (SE) blocks, which enhance feature representation by modeling

channel interdependencies through an attention mechanism. The SE block, as introduced by Hu et al. (2018), adaptively recalibrates channel-wise feature responses to emphasize informative features while suppressing less useful ones.

The architecture consists of three convolutional blocks, each followed by batch normalization, ReLU activation, and an SE block. The SE block performs two key operations:

- **Squeeze**: Global average pooling across spatial dimensions to generate channel-wise statistics
- **Excitation**: A fully connected layer that produces modulation weights for each channel

This channel-wise attention mechanism has been shown to improve model performance with minimal computational overhead (Hu et al., 2018). The SE blocks in our implementation use a reduction ratio of 16, balancing parameter efficiency and representational power.

2.5.2 Custom CNN-Specific Training Approach

Unlike the transfer learning approaches used with pre-trained models, our custom CNN was trained from scratch with some specific optimization strategies:

- **Cosine Annealing scheduler**: Our learning rate schedule follows a cosine annealing pattern with a period of 10 epochs, allowing the learning rate to oscillate and potentially escape local minima (Loshchilov and Hutter, 2017).
- **Specialized augmentation**: The custom CNN particularly benefited from more aggressive data augmentation strategies to compensate for the lack of pre-trained weights, including stronger rotations and more extensive color jittering than used with the transfer learning models.

Algorithm 1 VGG16Modified Architecture

function VGG16MODIFIED
    Load pre-trained VGG-16 with ImageNet weights
    Extract number of features from final layer: $num\_ftrs \leftarrow$ VGG.classifier[6].in_features
    Replace final classifier layer with:
    Dropout(p=0.4)
    Linear($num\_ftrs \rightarrow 2$) ▷ Binary classification
end function
function FORWARD(x)
    return VGG(x)
end function

Algorithm 2 ViTModified Architecture

function VITMODIFIED
    Load pre-trained ViT (vit_base_patch16_224) with ImageNet weights
    Extract number of features: $num\_ftrs \leftarrow$ ViT.head.in_features
    Replace classification head: ViT.head $\leftarrow$ Linear($num\_ftrs \rightarrow 2$)
end function
function FORWARD(x)
    return ViT(x)
end function

Algorithm 3 EnhancedViT Architecture

function ENHANCEDVIT(dropout_rate=0.3, hidden_dim=512)
    Load pre-trained ViT (vit_base_patch16_224) with ImageNet weights
    Remove classification head: ViT.head $\leftarrow$ Identity()
    Extract embedding dimension: $embed\_dim \leftarrow$ ViT.embed_dim
    Create attention layer:
    Linear($embed\_dim \rightarrow 1$)
    Create classifier:
    LayerNorm($embed\_dim$)
    Dropout($dropout\_rate$)
    Linear($embed\_dim \rightarrow hidden\_dim$)
    ReLU()
    Dropout($dropout\_rate$)
    Linear($hidden\_dim \rightarrow 2$)
end function
function FORWARD(x)
    $tokens \leftarrow$ ViT.forward_features(x) ▷ Shape: [batch, num_tokens, embed_dim]
    $attn\_scores \leftarrow$ attention_layer($tokens$) ▷ Shape: [batch, num_tokens, 1]
    $attn\_weights \leftarrow$ Softmax($attn\_scores$, dim=1)
    $weighted\_feature \leftarrow$ Sum($attn\_weights \cdot tokens$, dim=1)
    $out \leftarrow$ classifier($weighted\_feature$)
    return out
end function

Algorithm 4 InterpretableViT Architecture

function INTERPRETABLEVIT(dropout_rate=0.3, hidden_dim=512)
    Load pre-trained ViT (vit_base_patch16_224) with ImageNet weights
    Remove classification head: ViT.head ← Identity()
    Extract embedding dimension: $embed\_dim \leftarrow$ ViT.embed_dim
    Create attention layer for patch tokens:
    Linear($embed\_dim \rightarrow 1$)
    Create classifier for combined representation:
    LayerNorm($embed\_dim \cdot 2$)
    Dropout($dropout\_rate$)
    Linear($embed\_dim \cdot 2 \rightarrow hidden\_dim$)
    ReLU()
    Dropout($dropout\_rate$)
    Linear($hidden\_dim \rightarrow 2$)
end function
function FORWARD(x)
    $tokens \leftarrow$ ViT.forward_features(x) ▷ Shape: [batch, N+1, embed_dim]
    $cls\_token \leftarrow tokens[:, 0, :]$ ▷ Extract CLS token
    $patch\_tokens \leftarrow tokens[:, 1:, :]$ ▷ Extract patch tokens
    $attn\_scores \leftarrow$ attention_layer($patch\_tokens$)
    $attn\_weights \leftarrow$ Softmax($attn\_scores$, dim=1)
    $weighted\_patch \leftarrow$ Sum($attn\_weights \cdot patch\_tokens$, dim=1)
    $combined \leftarrow$ Concatenate(cls_token, $weighted\_patch$, dim=1)
    $logits \leftarrow$ classifier($combined$)
    return $logits$, $attn\_weights$ ▷ Return both for interpretability
end function

Algorithm 5 InceptionV3Modified Architecture

function INCEPTIONV3MODIFIED
    Load pre-trained Inception v3 with ImageNet weights
    Extract number of features: $num\_ftrs \leftarrow$ Inception.fc.in_features
    Replace classification layer:
    Dropout(0.5)
    Linear($num\_ftrs \rightarrow 2$) ▷ Binary classification
end function
function FORWARD(x)
    if self.training **then**
        aux_out, $out \leftarrow$ Inception(x) ▷ Return auxiliary output during training
        return aux_out, out
    else
        return Inception(x)
    end if
end function

Algorithm 6 ResNet50Modified Architecture

function RESNET50MODIFIED
    Load pre-trained ResNet-50 with ImageNet weights
    Extract number of features: $num\_ftrs \leftarrow$ ResNet.fc.in_features
    Replace final fully connected layer:
    Dropout(0.5)
    Linear($num\_ftrs \rightarrow 2$) ▷ Binary classification
end function
function FORWARD(x)
    return ResNet(x)
end function

Algorithm 7 Custom CNN with Squeeze-and-Excitation Blocks

function SEBLOCK(channels, reduction=16)
    Create fully connected layers:
    fc1 $\leftarrow$ Linear($channels \rightarrow channels/reduction$)
    fc2 $\leftarrow$ Linear($channels/reduction \rightarrow channels$)
end function
function SEBLOCK.FORWARD(x)
    $batch, channels, , \leftarrow x$.size()
    $se \leftarrow$ Mean(x, dims=(2, 3)) ▷ Global Average Pooling
    $se \leftarrow$ ReLU(fc1(se))
    $se \leftarrow$ Sigmoid(fc2(se))
    $se \leftarrow se$.view($batch, channels, 1, 1$)
    return $x \cdot se$ ▷ Channel-wise multiplication
end function
function IMPROVEDCNN
    Create convolutional layers:
    Conv2d($3 \rightarrow 32$, kernel=3, stride=1, padding=1)
    BatchNorm2d(32)
    ReLU()
    SEBlock(32)
    MaxPool2d(kernel=2, stride=2)
    Conv2d($32 \rightarrow 64$, kernel=3, stride=1, padding=1)
    BatchNorm2d(64)
    ReLU()
    SEBlock(64)
    MaxPool2d(kernel=2, stride=2)
    Conv2d($64 \rightarrow 128$, kernel=3, stride=1, padding=1)
    BatchNorm2d(128)
    ReLU()
    SEBlock(128)
    MaxPool2d(kernel=2, stride=2)
    Create fully connected layers:
    Flatten()
    Linear($128 \cdot 16 \cdot 16 \rightarrow 512$)
    ReLU()
    Dropout(0.5)
    Linear($512 \rightarrow 2$) ▷ Binary classification
end function
function IMPROVEDCNN.FORWARD(x)
    $x \leftarrow$ conv_layers(x)
    $x \leftarrow$ fc_layers(x)
    return x
end function

Algorithm 8 Transfer Learning Framework for Fine-Grained Classification

```
function TRANSFERLEARNINGMODEL(model_name, num_classes=2)
    Load pre-trained model with ImageNet weights
    Extract model-specific features:
    Identify final layer dimensions
    Preserve feature extraction layers/blocks
    Replace classification head:
    Apply regularization (e.g., Dropout with appropriate rate)
    Create new classifier mapping to target classes
    Configure optimization strategy:
    Select appropriate optimizer (Adam/AdamW)
    Set learning rate and weight decay parameters
    Implement learning rate scheduler
    Define data augmentation pipeline:
    Apply standard transformations (resize, normalize)
    Implement model-specific augmentations
    Return modified model
end function
function MODELFORWARD(input)
    Extract features from modified backbone
    Apply classification head
    Return predictions
end function
function TRAININGLOOP(model, train_loader, val_loader, epochs)
    for epoch in 1 to epochs do
        Train model on training data
        Evaluate on validation data
        Update learning rate based on scheduler
        Save best model based on validation metrics
    end for
end function
```