# VLG Recruitment Challenge ‘24 Report

- **Abhiraj Bharangar (24117002) (9119079453)**

## Introduction

The objective of the VLG Recruitment Challenge was to classify animal species into 40 distinct categories. The dataset consisted of images from various classes, with significant diversity in quality, lighting, orientation, and labeling consistency. To tackle this task, we employed **ConvNeXt-Tiny**, a state-of-the-art architecture derived from transformer-inspired principles while retaining convolutional roots. The model was fine-tuned on the dataset with extensive data augmentation, class balancing techniques, and regularization strategies to mitigate overfitting and improve generalization.Some other preferable models which i tested were GoogleNet(InceptionV3),VGG 16,Resnet 52,102,Conv-next-base,Efficient Net and conv-next-small **but got the best result on conv-next Tiny.**

This report details the challenges faced, the methodologies adopted, and the insights gained during the model development process. It concludes with an explainability section to demystify the decision-making process of the ConvNeXt-Tiny model.

**Architecture**:

- Lightweight model with ~28M parameters, designed for efficiency and scalability.
- Composed of **4 hierarchical stages** with progressively larger feature dimensions.

**Innovations**:

- **Large Kernels**: 7x7 convolutions for capturing broader context.
- **Depthwise Convolutions**: Efficient channel-wise operations for spatial features.
- **Layer Normalization (LN)**: Replaces BatchNorm for simplicity and improved training.

**Training Optimizations**:

- **AdamW Optimizer** and **Cosine Annealing Scheduler** for efficient convergence.
- Compatible with advanced augmentations like **MixUp**, **CutMix**, and **Random Erasing**.

**Performance**:

- Competitive with state-of-the-art models while being computationally efficient (~4.5 GFLOPs for a 224x224 image).
- Scalable for larger datasets with options to move to ConvNeXt-Base or Large.
- In my case it was noticed due to lot of overfittings and less amount of data **ConvNeXt-Tiny was the most preferable choice**

## Dataset Analysis

### 1. Dataset Composition

- **Training Set:** Contained images divided into 40 subfolders (classes), each representing an animal species. Each class had varied numbers of images, ranging from as low as 150 to as high as 250
- **Test Set:** Consisted of 3,000 unlabeled images to evaluate the model's generalization.

### 2. Dataset Challenges

- **Class Imbalance:**
  Some classes had significantly fewer samples, leading to a bias in model predictions toward overrepresented classes.
  **Solution:** Weighted random sampling and synthetic augmentation techniques (e.g., **CutMix**) can be used or in my case **I augmented each class to have the same number of images upon which the model will train on**.
- **Redundancies and Visual Similarity:**
  Some classes contained duplicate or near-duplicate images, while others exhibited high intra-class variance.
  **Solution:** Augmentations ensured the model trained on diverse visual representations.

---

## Model Development

### 1. Preprocessing and Augmentation

- **Base Transformations:**
  Images were resized to 224×224, normalized, and converted to tensors.

```
base_transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize to 224x224 for ConvNeXt-Small input
    transforms.RandomHorizontalFlip(),  # Random horizontal flipping for data augmentation
    transforms.RandomRotation(30),  # Random rotation to simulate different orientations
    transforms.ToTensor(),  # Convert PIL image to a PyTorch tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # Normalize with ImageNet stats
])
```

- **Augmentation Techniques:**
  - **Random Horizontal Flips and Rotations** simulated natural variations.
  - **Color Jitter** altered brightness, contrast, and saturation to handle lighting differences. Note-> Synthetic augmentations like **Cut-Mix** could be particularly useful for this dataset and number of images per class is small and a big challenge for this dataset for **overfitting by a huge margin and in such cases synthetic augmentations like Cutmix can be particularly useful**

**2. ConvNeXt-Tiny Architecture**

## 1. Overall Structure:

- The model follows a hierarchical design like ResNet but incorporates modern advancements inspired by vision transformers.
- The input passes through a **Patchify Stem** and is then processed by four stages of **ConvNeXt Blocks** with progressively larger feature dimensions.
- At each stage, the feature map size is reduced using **downsampling layers**, and the channel dimensions are increased.

---

## 2. Patchify Stem:

- **Goal**: Prepare the input by downsampling it and converting it into patches.
- **Components**:
  - **Conv2D**: A large 4x4 convolution with a stride of 4, effectively reducing the spatial resolution by a factor of 4 (e.g., 224x224 → 56x56).
  - **Layer Normalization (LN)**: Replaces traditional BatchNorm to normalize feature maps.

---

## 3. Stages:

ConvNeXt is divided into 4 stages, each consisting of several ConvNeXt Blocks. The number of blocks and feature dimensions increase with depth.

- **Stage 1**:
  - Input resolution: 56x56 (downsampled from 224x224 in the Patchify Stem).
  - Number of ConvNeXt Blocks: 3.
  - Feature dimensions (number of channels): 96.
- **Stage 2**:
  - Input resolution: 28x28.
  - Number of ConvNeXt Blocks: 3.
  - Feature dimension: 192.
- **Stage 3**:
  - Input resolution: 14x14.
  - Number of ConvNeXt Blocks: 9 (deeper stage for richer feature extraction).
  - Feature dimensions: 384.
- **Stage 4**:
  - Input resolution: 7x7.
  - Number of ConvNeXt Blocks: 3.

    - Feature dimensions: 768.

Each stage progressively reduces the spatial dimensions (via downsampling layers) and increases the channel dimensions to capture finer details and more abstract features.

---

### 4. ConvNeXt Block:

Each ConvNeXt block incorporates the following:

- **Depthwise Convolution (7x7)**:
    - Operates spatially while keeping the channels independent, reducing computational costs.
- **Layer Normalization (LN)**:
    - Normalizes feature maps for stable training.
- **Pointwise Convolution (1x1)**:
    - Performs a channel mixing operation.
- **GELU Activation**:
    - Introduces smooth non-linearity for better gradient flow.
- **Linear Layer**:
    - Expands and contracts the channels for feature transformation (e.g., from 4x expansion).

---

### 5. Downsampling Layers:

- Used at the beginning of each stage (except Stage 1).
- **Conv2D with 2x2 Kernel and Stride 2**:
    - Reduces the spatial resolution by half while doubling the channel dimensions.

---

### 6. Classifier Head:

- **Global Average Pooling (GAP)**:
    - Reduces the spatial dimensions to a single vector by averaging each feature map.
- **Fully Connected (Linear)**:
    - Projects the pooled features to the desired number of output classes.
- **Dropout**:
    - Prevents overfitting by randomly dropping neurons during training.

---

### 7. Parameters and FLOPs:

- **Total Parameters**: ~28 million.
- **FLOPs**: ~4.5 GFLOPs (for 224x224 input).

---

### 8. Advantages:

- Combines **CNN simplicity** with modern training techniques.
- Scalable across Tiny, Base, and large versions.
- Suitable for a wide range of tasks with minimal modification.

.

## Model Customization and Regularization

### Fine-Tuning and Custom Head

- The pre-trained ConvNeXt-Tiny was fine-tuned by unfreezing certain layers to adapt to the dataset:
    - **Last Two Blocks:** Unfrozen to allow adaptation of mid- and high-level features.
    - **Custom Classification Head:** Replaced the default head with:
        - **Dropout (0.5):** Prevent overfitting.
        - **Fully Connected Layers (512 neurons):** Added non-linearity and feature refinement.
        - **Batch Normalization:** Improved training stability and convergence.

### Loss Function

- **CrossEntropy Loss with Label Smoothing (factor = 0.2):**
  Reduced overfitting by softening target probabilities, ensuring the model does not predict incorrect classes.

### Optimizer and Learning Rate Scheduler

- **AdamW Optimizer:** Combines the benefits of Adam and weight decay for better generalization.
- **Cosine Annealing Scheduler:** Reduced the learning rate smoothly to avoid abrupt drops, improving convergence over 20 epochs.

## Regularization Techniques

- **Dropout Layers:** Applied in multiple layers, including the classification head, to reduce reliance on specific features.
- **Weight Decay:** A regularization term (1×10−31 \times 10^{-3}1×10−3) was added to penalize large weights.

## Training Setup->

- ***Preprocessing->***
- Data Augmentation was performed after resizing image to 224x224x3 as mentioned above using various augmentation techniques like rotation (30) horizontal flipping etc. And then image was normalized and resized using standard image-net metrics
- Weighted samplers were used so that imbalances between classes were handled (one could also augment each class having same number of images e.g. Say 1000 images per class)
- Later in the code a validation set using 15% of the training data was created to monitor validation set loss, so that if it increased 3 times straight the program was terminated from training more epochs and best model gets saved

---

## OTHER PROCESSING STEPS->

- Subprocesses to use for data loading(num_workers)=4
- A custom head was used
- self.model.classifier[2] = nn.Sequential(

```
    nn.Dropout(0.6),
    nn.Linear(in_features, 512),
    nn.BatchNorm1d(512),
    nn.ReLU(),
    nn.Dropout(0.6),
    nn.Linear(512, num_classes))
```

∉ **Hardware:** NVIDIA RTX 4050 GPU with Automatic Mixed Precision (AMP) enabled for faster computation.
- **Batch Size:** Set to 32 to balance GPU memory constraints and training stability. (Later was tried with 64 also in any chance of reducing overfitting)
- **Early Stopping:** Training was halted if validation loss did not improve for 3 consecutive epochs.
- **Loss Function: Cross** Entropy loss with label smoothing was used (0.2)
- **Optimizer**:AdamW optimizer was primarily used with weight decay of 1e-3(due to a lot of overfittings in the previous models tested on test set)
- Scheduler:Cosine annealing learning rate scheduler was used (with T_max=20) and initial learning rate was set to 0.001

## Model Explainability

### 1. Feature Hierarchy

- **Low-Level Features:** The model first learns basic patterns such as edges and textures.
- **Mid-Level Features:** Recognizes body parts like legs, wings, and tails.
- **High-Level Features:** Identifies full objects and distinguishes between species based on distinctive features.

### 2. Grad-CAM Analysis

Grad-CAM visualizations revealed:

- Focus on species-specific features such as fur texture, color patterns, and facial structures.
- Misclassifications primarily arose in visually similar species, e.g., different breeds of cats or birds with similar plumage.

---

## Results and Observations

### Training and Validation Metrics

- **Validation Accuracy:** Peaked at 96% (side note->clearly shows even with excess of dropout, weight decay parameters the overfitting %, as there is enormous difference in validation and test accuracy, so something is different in the test set than the training set as validation set was created from the training set only)
- **Training Accuracy:** Peaked at 97.8%.

### Test Set Performance

- Achieved **59.1% accuracy** on the test set (private leaderboard) and about 58% on public leaderboard.

## Challenges Faced

- ***Overfitting:***
  Observed large overfitting despite dropout and regularization, especially in larger models. Clearly implying either there was noise in the data (which did not seem to be the case), Rather there were many other unidentified classes which were in the test set but not available for the model to train on as they were not present on the training set which would clearly explain the following points->

- *The validation accuracy was quite high as well as the training accuracy on the final epochs but there was a* ***stark difference in validation and testing accuracy since validation set was made from using 15% of the training data which also did not***

***include the several classes which were potentially nonexistent in the training data and hence my model gave such a low accuracy on the testing data****. This could have been tackled by* ***zero-shot learning models*** *like* ***CLIP library by OPENA****I (note-> This observation was noted by me only near the end of the competition and i could not implement this, if had implemented this while keeping the rest of the things same this could have given easily a* ***20%+ boost on accuracy on testing set****)*

Some general techniques used to reduce overfitting (only gave me a small boost on testing data)
**Addressed via:** Label smoothing, increase in weight decay and dropouts, and class-aware sampling.

- **Data Quality:**
  Mislabels and redundancies affected generalization.
- **Hardware Limitations:**
  Adjusted batch sizes and epochs to fit the RTX 4050 GPU memory.

### Insights Gained

- Pretrained models like ConvNeXt-Tiny adapt well to small datasets but require careful regularization.
- Label smoothing and Cut Mix significantly improved robustness to noisy labels.
- Test performance could be further improved by refining the dataset and employing k-fold cross-validation.

## Conclusion and Future Work

**Key Takeaways->**

- The most important aspect of the model was to understand how even after creating a validation set one could see a stark difference between accuracy scores on validation set and testing data. This clearly implied that there was something different about the training data and testing data which was that **several unidentified classes were present in testing data which were not a part of the training set** hence i**mplementation of zero shot learning models** would have given a huge performance boost.

- **ConvNeXt-Tiny** proved effective for animal classification but struggled with subtle inter-class variations.
- Regularization techniques like dropout and label smoothing mitigated overfitting but did not eliminate it entirely.

### Future Directions

- *Apply zero-shot learning algorithms or like the CLIP learning library from OPEN AI to get huge performance boost. (could not implement this due to time constraints realized this only on the last day of competition)*

1. **Ensemble Learning:** Combine multiple models for better generalization.
2. **Data Augmentation:** Explore GANs or advanced augmentation techniques to generate synthetic data.
3. **Explainability:** Incorporate SHAP or LIME to enhance model interpretability.
4. **Hyperparameter Tuning:** Use tools like Optuna to optimize learning rate, dropout, and weight decay. (Optuna could boost performance by 1-3% very easily many times)