Fine-Grained Image Classification

Computer vision report

Lopez Calvo Joaquin
joaquin.lopezcalvo@studenti.unitn.it
University of Trento
Trento, Italy

Marconi Sonia
sonia.marconi@studenti.unitn.it
University of Trento
Trento, Italy

Orlandi Damiano
damiano.orlandi@studenti.unitn.it
University of Trento
Trento, Italy

1 ABSTRACT

This work is developed in the context of a competitive machine learning challenge on fine-grained image classification, which emphasizes not only predictive performance but also methodological rigor, reproducibility, and generalization under limited labeled data. Participants are provided with an initial training set and are allowed to enrich it through external data collection, encouraging both data-centric and model-centric approaches.

We compare convolutional neural networks (CNNs) and Vision Transformers (ViTs), with a primary focus on ViT-based models due to their strong performance during the competition. Experiments are conducted on multiple fine-grained datasets, including mammals, flowers, aircraft, and cars, and performance is assessed using standard top-1 and top-5 accuracy metrics on a held-out test set released only at evaluation time, ensuring an unbiased comparison.

Results show that two ViT-based models achieve an accuracy of approximately 72% on the competition dataset of mammal species, outperforming alternative approaches. While the study is limited by the exploration of a restricted set of datasets, models, and hyperparameter configurations, it demonstrates the effectiveness of Vision Transformers for fine-grained image classification and provides insights into the impact of design choices in data preparation and model optimization.

2 INTRODUCTION

Fine-Grained Image Classification (FGIC) is a challenging subfield of computer vision that focuses on distinguishing between visually similar subcategories within a broader semantic class. Unlike coarse-grained classification, where categories differ by clearly identifiable visual traits, FGIC requires the recognition of subtle and localized visual cues, often under significant variations in pose, illumination, background, and scale. Closely related to FGIC, Fine-Grained Image Analysis (FGIA) lies between basic-level category recognition and instance-level analysis, addressing classification problems where objects belong to the same meta-category and exhibit high visual similarity. The intrinsic difficulty of FGIC and FGIA arises from the combination of small inter-class variance and large intra-class variance. Models must therefore learn discriminative representations capable of capturing fine visual details while remaining robust to contextual and appearance changes. These challenges are particularly evident in biological and ecological domains, where species-level classification is required and visual differences between classes may be minimal. Applications include wildlife monitoring, biodiversity assessment, ecological surveillance, and biological research, where accurate fine-grained recognition can provide valuable scientific insights. Recent advances in deep learning have significantly improved performance in fine-grained recognition tasks. In particular, transformer based architectures have shown strong potential due to their ability to model long-range dependencies and global contextual relationships through attention mechanisms. Vision Transformers (ViTs) have emerged as a competitive alternative to convolutional neural networks, especially in scenarios where capturing global structure and subtle feature interactions is critical.

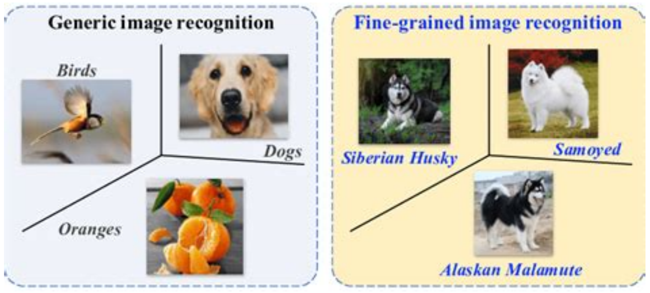

Figure 1: Distinction between an image recognition and a fine-grained task

In this work, we address the problem of fine-grained mammal image classification by proposing an end-to-end learning pipeline that integrates data preparation, augmentation, representation learning, and classification within a unified framework. The approach is designed to operate effectively in limited-data regimes, which are common in fine-grained visual tasks, and emphasizes robustness and generalization rather than reliance on large-scale labeled datasets. Data augmentation strategies are employed to mitigate data scarcity, while transformer-based models are leveraged to extract expressive visual representations.

The main contributions of this paper are threefold. First, we provide a systematic formulation of a fine-grained image classification pipeline tailored to biologically motivated visual categories. Second, we investigate the effectiveness of Vision Transformer-based architectures for fine-grained recognition under constrained data conditions. Third, we offer an empirical analysis of model behavior, highlighting the role of data handling and architectural choices in achieving competitive performance on fine-grained classification tasks.

3 RELATED WORK

In this section, we provide an overview of the techniques, methods, and support tools utilized in the preparation of this report.

3.1 Approaches to Fine-Grained Image Recognition

3.1.1 Recognition by Localization-Classification Subnetworks. According to [3] models in this category typically involve:

- **Detection or Segmentation Techniques**: Employing methods to locate key image regions corresponding to fine-grained object parts, enhancing the discriminative capabilities of the classification sub-network.
- **Deep Filters**: Utilizing deep convolutional neural networks (CNNs) which employ deep filters to capture localized descriptors linked to semantic parts of objects.
- **Attention Mechanisms**: Integrating attention mechanisms in CNNs to focus on loosely defined regions for fine-grained objects, enhancing accuracy by selectively attending to salient parts.
- **Spatial Transformers Networks (STN) and Feedback Mechanisms**: Approaches that improve accuracy by enabling spatial transformations or using reinforcement learning to capture representations at multiple granularities.

3.1.2 End-to-End Feature Encoding. This approach simplifies the pipeline by encoding features through deep networks without explicitly detecting or segmenting parts, reducing dependency on heavily annotated datasets.

3.1.3 Recognition with External Information. Enhancing the recognition process by leveraging additional data sources or annotations that provide context or supplementary details aiding in the distinction between closely related categories.

3.2 Transformers

Transformers are a neural network architecture introduced in the paper "Attention is All You Need" by Vaswani et al. [26]. It revolutionized the field of natural language processing by improving state-of-the-art performances on various tasks with a design that overcome some of the limitations of the sequential logic behind Recurrent Neural Networks. The Transformers architecture relies on self-attention mechanisms to capture dependencies between input elements and has been widely adopted in sequence modeling tasks such as machine translation, text generation, and sentiment analysis [2].

3.2.1 Key Components. The Transformers architecture consists of two main components:

- **Encoder**: Processes the input sequence and generates a context representation for each input token.
- **Decoder**: Generates the output sequence based on the context representations produced by the encoder.

3.2.2 Attention. These mechanisms play a crucial role in Transformers architectures, allowing models to focus on relevant parts of the input sequence when processing it. By attending to different parts of the input, the model can capture long-range dependencies and relationships between elements more effectively. Attention mechanisms enable the model to focus on relevant parts of the input sequence and capture complex relationships between elements, leading to improved performance in various NLP and sequence modeling tasks.

3.2.3 Multi-Head Attention. It is a variant of the attention mechanism used in Transformers. In multi-head attention, the input is transformed into multiple representations (heads) by applying linear projections. Each head computes attention independently, allowing the model to capture different aspects or types of relationships in the input sequence. The outputs of the different heads are concatenated and linearly transformed to obtain the final output.

3.2.4 Self-Attention Mechanism. The core of the Transformers architecture is the self-attention mechanism, which allows the model to weigh the importance of different input tokens when processing each token. Self-attention computes attention scores between each pair of tokens in the input sequence and generates context representations by aggregating information from relevant tokens.

3.2.5 Cross-Attention Mechanism. In addition to self-attention, the decoder utilizes a cross-attention mechanism. This mechanism allows the decoder to focus on relevant parts of the input sequence (processed by the encoder) while generating each token of the output sequence. Cross-attention aligns the encoder's context representations with the decoder's generation process, ensuring accurate and coherent output.

3.2.6 Special Tokens. In Transformers, special tokens such as the classification token are used to indicate the task at hand. For instance, in text classification tasks, a classification token is added to the input sequence to signal that the model needs to classify the text into specific categories. This token serves as the focal point for processing the input sequence in the model.

In conclusion, the self-attention and cross-attention mechanisms, along with linear projections in Transformers, have set new benchmarks across various NLP tasks by effectively capturing long-range dependencies and relationships between input elements. In the following section we will see how these advances were capitalized for computer vision as well.

3.3 Data augmentation

Data augmentation is an essential component of the data preparation pipeline in machine learning, particularly for tasks involving image classification. This technique involves generating new training examples from existing ones by applying various transformations to the images. These transformations can include rotations, translations, flips, crops, adjustments to brightness and contrast, and more. While seemingly straightforward, data augmentation plays a critical role in enhancing the diversity and size of training datasets, which is crucial for building more robust and generalizable deep learning models.

3.3.1 Importance of Data Augmentation. In general, data augmentation is particularly valuable because collecting new data can be challenging and expensive. By artificially expanding the dataset, data augmentation introduces variability, which helps in training deep learning models that are less likely to overfit. Overfitting occurs when a model learns the training data too well, including the noise and fluctuations, to the detriment of its performance on new, unseen data. By using augmented data, the model learns to generalize from a broader set of input variations, thereby improving its accuracy and robustness on unseen data.

3.3.2 *Application and Considerations.*

In the context of image classification, it's crucial to apply transformations that do not alter the inherent classification of an image. For instance, a cat in an image remains a cat regardless of the image's orientation or lighting conditions. However, care must be taken that the transformations do not mislead the training process. For example, if an image is cropped in a way that the main subject (e.g., a cat) is no longer visible, the original label becomes incorrect, leading to potential training issues.

The process of choosing specific transformations often depends on the characteristics of the dataset and the task at hand. Moreover, while augmenting the data, it is common practice to only apply these transformations to the training set and not the validation set; applying augmentation to the validation set could distort the natural distribution and characteristics of the data, potentially leading to misleading results about the model's performance. In fact, the validation set is used to evaluate how well the model generalizes to new data that it has not seen during training. It acts as a proxy for real-world data. By keeping the validation set free of augmentation, we ensure that it remains a reliable benchmark for testing the model's performance under normal conditions that are expected in practical applications.

3.3.3 *Implementation.*

The following sources guided our implementation of data augmentation strategies: [5], [15].

3.3.4 *Data Augmentation.*

Considering the reduced size of the dataset, we made the decision to increase its size consistently by implementing many functions contained in the v2 library, an optimized version of the transform one, to enhance the model's ability to generalize from the training data.

```
def get_transforms():
    return v2.Compose([
        v2.Pad(10),
        v2.Resize(256),
        v2.RandomCrop(size=(224, 224)),
        v2.RandomHorizontalFlip(p=0.5),
        v2.RandomRotation(15),
        v2.ColorJitter(
                brightness=0.2,# bad picture conditions
                contrast=0.2, # underwater images
                saturation=0.2,),
        v2.ToTensor(),
        v2.Normalize(mean=[0.485, 0.456, 0.406],
                     std=[0.229, 0.224, 0.225]),])
```

The Python function *get_transforms()* defines a sequence of image preprocessing steps for data augmentation and normalization, operating both on the geometrical augmentation and on the color one. These are the functions applied:

Padding: *v2.Pad(10)*: This applies padding of 10 pixels to all sides of the image in order to maintain spatial information near the borders.

Resizing and Cropping: *v2.Resize(256)*: Resizes the image to 256x256 pixels to ensure that all images fed into the model have the same dimensions. *v2.RandomResizedCrop(size=(224, 224))*: Performs a crop of the image at a random location and resizes the crop to 224x224 pixels. This transformation is essential for models that require a fixed input size. It also introduces variability in the part of the image seen by the model during training, enhancing generalization.

Random Horizontal Flip: *v2.RandomHorizontalFlip(p=0.5)*: Flips the image horizontally with a probability of 0.5. This augmentation mimics the variability in real-world scenarios where objects of interest may be oriented differently.

Random Vertical Flip: *v2.RandomVerticalFlip(p=0.15)*: Flips the image vertically with a probability of 0.15. With regard to the potential images of the competition dataset, like aircrafts, birds, and flowers, we included this function with a low percentage to simulate the training during the competition day.

Color Jitter: *v2.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2)*: Adjusts brightness, contrast, saturation, and hue by specified amounts. This helps the model cope with different lighting conditions and color variations in images, which can be particularly useful if the model is deployed in environments with varying lighting and exposure conditions.

Conversion and Normalization: *v2.ToTensor()*: Converts the PIL Image to a PyTorch tensor, which is a required transformation for image data when training with PyTorch models. *v2.Normalize (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])*: Normalizes the image tensor using mean and standard deviation, aligning the data distribution with what the pre-trained models typically expect. This step is crucial for models trained with normalized datasets like ImageNet, as it ensures that the input data is on a comparable scale during training. This is why we preferred not to compute the mean and standard deviation of the images in the dataset.

3.3.5 *Torchvision Transforms.v2.*

The `torchvision.transforms.v2` module, a part of the popular PyTorch framework, introduces a robust set of tools designed to enhance and simplify the process of image augmentation and transformation in machine learning. This updated module extends the functionality of its predecessor, `torchvision.transforms`, by introducing new capabilities that are tailored for advanced image processing challenges, especially those encountered in fine-grained and complex visual recognition tasks.

So, `torchvision.transforms.v2` is not just an incremental update but a significant overhaul designed to accommodate the growing needs of the computer vision community. This module includes:

- Enhanced support for a wider range of image types, including support for video frames and volumetric images, enabling comprehensive transformations across different media types.
- Improved handling of bounding boxes, masks, and other forms of annotations, which are crucial for tasks that involve object detection, segmentation, and other forms of detailed image analysis.

Moreover, the module incorporates several technical innovations that allow for more precise and context-aware transformations:

- **Context-Aware Cropping and Resizing**: Algorithms within this module can adapt cropping and resizing operations based on the content of the image, ensuring that important features within the image are preserved and emphasized during preprocessing.
- **Advanced Color Augmentation**: It offers sophisticated color adjustment capabilities that go beyond simple color transformations to include adjustments that can mimic different lighting conditions and enhance contrast and saturation to make the training data more diverse and robust against varying conditions.
- **Geometric Transformations**: Includes a variety of new geometric transformations that allow for more complex manipulations of images, such as rotations at arbitrary angles, skewing, and perspective adjustments, all of which can introduce a richer set of variations to model training.

In the context of fine-grained image recognition, *torchvision.transforms.v2* proves particularly valuable:

- It facilitates the extraction and emphasis of subtle visual differences between highly similar subcategories, which is essential for tasks like distinguishing between different species of birds or models of cars.
- The module's ability to handle detailed annotations allows for more effective training of models that rely on localized features within an image, improving the accuracy and reliability of the recognition tasks.

To summarize, the `torchvision.transforms.v2` module significantly enhances the toolkit available for developers and researchers in the field of machine learning and computer vision. By providing more detailed and adaptable image processing capabilities, this module supports the development of highly accurate and efficient visual recognition systems that are capable of handling the complexities of real-world visual data.

An example of the syntax is provided below:

Code 1 Torchvision Transformations.v2

```
Require: im_dimension, original_image
  import torchvision.transforms as v2
  data_transforms = {
      'train': v2.Compose([
          v2.Resize((244, 244)),
          v2.RandomRotation(15),
          v2.RandomCrop(224),
          v2.RandomHorizontalFlip(),
          v2.ToTensor(),
          v2.Normalize(mean, std)
      ])
  transformed_image = data_transforms['train'](original_image)
```

3.3.6 Conclusion. Effective data augmentation not only combats the risk of overfitting but also enriches the training data without the need for additional data collection. By understanding and implementing strategic transformations, developers can significantly improve the performance and reliability of their image classification models. For references, please refer to [4].

3.4 Activation functions

An activation function in a neural network is a mathematical gate in between the input feeding the current neuron and its output going to the next layer. These functions are critical as they decide whether a neuron should be activated or not, making the decision based on whether each neuron's input is relevant for the model's prediction. The activation function's purpose is to introduce non-linearity into the output of a neuron. This is important because most real-world data is non-linear, meaning it cannot be separated or classified without non-linear computations. Without activation functions, neural networks would essentially behave like a single-layer perceptron, because summing these linear transformations (no matter how many layers there are) still results in a linear transformation (references may be found in [**?**] and [7]). Let us explore the activation functions that were used for our purpose:

3.5 Learning rate

In machine learning, the **learning rate** is a fundamental hyperparameter that controls the size of the steps taken during the optimization process as the model trains. It determines how much the model's weights should be adjusted with respect to the loss gradient during each iteration of training. The learning rate is crucial for guiding how quickly a model can converge to a minimum of the loss function, balancing the speed and stability of the learning process.

When updating the weights in neural network training, the learning rate η plays a direct role in the formula used for adjustments:

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \nabla L(w_{\text{old}}) \tag{1}$$

where:

- w_{old} and w_{new} are the old and new weights, respectively,
- $\nabla L(w_{\text{old}})$ is the gradient of the loss function at the old weights.

A **higher learning rate** may lead to faster convergence but risks overshooting the minimum, potentially causing the training to diverge. Conversely, a **lower learning rate** ensures more stable convergence and a thorough exploration of the loss landscape, although it may slow down the training process and risk getting stuck in local minima.

Modern training techniques often involve **adaptive learning rate** methods, where the learning rate may change dynamically based on the training epoch or performance metrics, helping to optimize the training process further.

3.6 Learning rate scheduler

A learning rate scheduler in machine learning is a strategy used to adjust the learning rate during training, typically to improve the convergence rate of a model. It is essential because the learning rate is one of the most crucial hyperparameters in training neural networks, directly influencing the speed and stability of the learning process. So, a learning rate scheduler dynamically modifies the learning rate during training. The idea is to start with a higher learning rate to make substantial progress toward the loss function's minimum, and then decrease the learning rate to make smaller, more precise steps as it approaches the minimum. Into steps:

- **Modification of the Learning Rate**: - The scheduler adjusts the learning rate η based on a specific strategy (e.g., after a certain number of epochs, based on the model's performance, etc.).

- **Control of the Learning Speed**: - At the start of training, a high learning rate helps make significant progress. Gradually reducing the learning rate allows the model to fine-tune its parameters precisely without making changes too large that might cause the model to miss the global minimum.

3.6.1 Examples of Schedulers.

- StepLR: it reduces the learning rate by a fixed factor after a certain number of epochs.

Code 2 StepLR

```
Require: model
  from torch.optim.lr_scheduler import StepLR
  scheduler = StepLR(optimizer, step_size=10, $gamma = 0.1$)
```

- ReduceLRonPlateau: it reduces the learning rate when a specific metric stops improving.

Code 3 ReduceLROnPlateau

```
Require: model
  from torch.optim.lr_scheduler import ReduceLROnPlateau
  scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=3)
```

- CosineAnnealingLR: it changes the learning rate following a cosine function, decreasing and then increasing again.

Code 4 CosineAnnealingLR Scheduler Setup

```
Require: model
  from torch.optim.lr_scheduler import CosineAnnealingLR
  scheduler = CosineAnnealingLR(optimizer, $T_max = 50$)
```

- CosineAnnealingLR Scheduler with Warmup : the *get_cosine_schedule_with_warmup* function from the *transformers* library is a learning rate scheduler that combines a warmup phase with a cosine annealing schedule. This approach can improve neural network training by managing the learning rate effectively throughout the training process. function from the *transformers* library is a learning rate scheduler that combines a warmup phase with a cosine annealing schedule. This approach can improve neural network training by managing the learning rate effectively throughout the training process.
  - Warmup
    - Initial Phase: At the start of training, a high learning rate can cause divergence. A warmup phase is introduced to mitigate this.
    - Linear Increase: During the warmup, the learning rate starts from a small value and increases linearly. This allows the model to adjust and begin learning effectively.
    - Warmup Steps:The number of warmup steps is predefined. For instance, setting 10% of total training steps as warmup steps means the learning rate will increase linearly for those initial steps.
  - Cosine Annealing
    - Post-Warmup Phase: After warmup, the learning rate follows a cosine annealing pattern.
    - Cosine Decay: The learning rate decreases following a cosine curve, which helps in fine-tuning the model's parameters.
    - Periodicity: The cosine function allows the learning rate to oscillate, which can help the model escape local minima and find better

Code 5 Training Loop with CosineAnnealingLR Scheduler with Warmup

```
Input: model, train_loader, criterion
Parameters: num_epochs = 30
num_training_steps = num_epochs * len(train_loader)
num_warmup_steps = int(0.1 * num_training_steps)
scheduler = get_cosine_schedule_with_warmup(optimizer, num_warmup_steps=num_warmup_steps, num_training_steps=num_training_steps)
for epoch in range(num_epochs) do
    model.train()
    for inputs, labels in train_loader do
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
    end for
    # Update the scheduler
    scheduler.step()
    print(f"Epoch {epoch+1}, Learning Rate: {scheduler.get_last_lr()}")
end for
```

Ultimately, the learning rate scheduler allows for dynamic adjustment of the learning rate, thereby improving the model's performance.

3.7 Optimizer

An **optimizer** is an algorithm used in machine learning and deep learning to update the weights of the model in order to minimize the loss function. The primary goal of the optimizer is to find the optimal parameters that reduce the prediction error of the model. Some of the most common optimizers include:

- **Stochastic Gradient Descent (SGD)**: A simple optimizer that updates the weights in the direction of the negative gradient of the loss function.
- **Adam (Adaptive Moment Estimation)**: an optimization algorithm designed to improve the speed and performance of the training process by adjusting the learning rate based on the data's characteristics and the gradients of the cost function.
- **AdamW**: a variant of the Adam algorithm. This one was the optimizer in the competition's model.

3.7.1 *AdamW.*

AdamW is a variant of the Adam algorithm introduced to improve regularization through weight decay. In Adam, L2 regularization (also known as weight decay) is applied only indirectly, which can lead to less effective regularization. AdamW modifies this approach by separating weight decay from the gradient updates. Let us focus on the characteristics of Adamw:

- **Separation of Weight Decay**: In Adam, the weight decay term is added to the gradients, which can cause less effective regularization. AdamW, on the other hand, applies weight decay separately.
- **Parameter Updates**: AdamW updates the parameters following the standard Adam update but adds a separate step to reduce the weights proportionally to the specified decay rate.
- **Advantages**:
  - **Improved Regularization**: The separation of weight decay improves regularization, reducing overfitting.
  - **Performance**: AdamW tends to provide better overall performance compared to Adam, especially in scenarios with complex models and large datasets.
- **Update Formula in AdamW**: Given a parameter θ with learning rate α, weight decay rate λ, first moment estimate m, and second moment estimate v:

(1) Calculation of moment estimates (as in Adam):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad (2)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \quad (3)$$

(2) Bias correction (as in Adam):

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad (4)$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad (5)$$

(3) Parameter update (as in Adam but with separate weight decay):

$$\theta_{t+1} = \theta_t - \alpha \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \theta_t \right) \quad (6)$$

Where:
- g_t is the gradient of the loss function with respect to the parameter θ.
- β_1 and β_2 are the exponential decay rates for the moment estimates.
- ϵ is a small constant to avoid division by zero.

In summary, AdamW is an improved version of Adam that applies more effective regularization through a separate weight decay method, enhancing model performance and reducing overfitting. For references, please refer to [**?**]

3.8 SpinalNet architecture

SpinalNet is a neural network architecture designed to improve performance and efficiency in deep learning tasks. The architecture is inspired by the structure and function of the human spinal cord, which processes sensory input in a hierarchical and segmented manner. This method aims to enhance gradient flow and reduce the vanishing gradient problem, common in deep networks.

3.8.1 *Key Features of SpinalNet.*

- **Segmented Processing**:
  - The input features are divided into segments, with each segment processed by different layers independently.
  - This segmented approach allows the network to handle large-dimensional data more effectively by breaking it down into manageable parts.
- **Hierarchical Integration**:
  - Outputs from each segment are progressively integrated, combining information from different layers.
  - This hierarchical combination allows for deeper and more refined feature extraction, improving the network's ability to learn complex patterns.
- **Improved Gradient Flow**:
  - By processing and combining segments in stages, SpinalNet enhances gradient flow throughout the network.
  - This reduces the vanishing gradient problem, allowing for more efficient training of deep networks.
- **Effective Regularization**:
  - The architecture incorporates dropout and batch normalization layers to prevent overfitting and stabilize training.
  - These techniques are crucial for maintaining model performance on unseen data.

3.8.2 *General Architecture.*

A typical SpinalNet consists of multiple spinal layers, where each layer processes a portion of the input features. The processed features are then concatenated and passed to the next layer. This process continues until all segments are processed and combined. The final concatenated output is then passed through a fully connected layer to produce the final predictions.

3.8.3 *Advantages of SpinalNet.*

- **Efficiency**: By dividing input features into segments and processing them hierarchically, SpinalNet can handle large-dimensional data more efficiently.
- **Performance**: The hierarchical integration of features allows for deeper feature extraction, improving the model's ability to learn and generalize.
- **Stability**: The use of dropout and batch normalization layers helps prevent overfitting and stabilize the training process.

To sum up, SpinalNet is an advanced neural network architecture that mimics the hierarchical and segmented processing of the human spinal cord. By dividing and progressively combining feature segments, it improves gradient flow, enhances feature extraction, and provides effective regularization. This makes SpinalNet a powerful tool for improving the performance and efficiency of deep learning models.

4 METHODOLOGY

4.1 Convolutional Neural Networks

The Residual Network (ResNet) represents a pivotal advancement in deep learning applied to computer vision. By integrating residual connections, or shortcuts, into its architecture, ResNet addresses the common challenges of vanishing gradients and performance degradation that typically arise as network depth increases. These skip connections act like bridges, allowing gradients to bypass layers during backpropagation, which helps in preserving the strength of the gradient and allows the network to learn faster and more effectively.

ResNet has revolutionized the field of CNNs by enabling the construction of much deeper networks without the usual pitfalls of deep networks. This is achieved through its unique ability to preserve input information across layers via skip connections, which maintain performance stability even as network depth increases.

Key Components of ResNet. The architecture of ResNet can be detailed through the following key components:

(1) **Initial Setup:**
  - **Conv2d:** The first convolutional layer (conv1) with a kernel size of 7×7, stride of 2, and padding of 3, processes 3-channel input images (RGB) and produces 64 feature maps. This layer helps in extracting low-level features like edges and textures from the images.
  - **BatchNorm2d:** Following the convolutional layer, batch normalization (bn1) is applied to stabilize and speed up the training. It normalizes the output of the previous layer, reducing internal covariate shift.
  - **ReLU:** The ReLU activation function is used for introducing non-linearity to the model, allowing it to learn more complex patterns.
  - **MaxPool2d:** A max pooling layer with a kernel size of 3 and stride of 2 reduces the spatial dimensions (height and width) of the output from the previous layer, thus reducing the computation for the upcoming layers and also helping in extracting dominant features which are rotational and positional invariant.

(2) **Residual Blocks:**
  - **Sequential:** The model includes multiple sequential blocks, each containing a series of bottleneck blocks. These are specially designed to make the network deeper without increasing the number of parameters significantly.
  - **Bottleneck:** Each bottleneck block in the layer1 and subsequent layers includes:
  (a) **Conv2d and BatchNorm2d:** Three convolutional layers each followed by batch normalization. The first and third convolutions typically use 1×1 kernels, primarily for dimensionality reduction and restoration, respectively, while the second uses a 3×3 kernel for processing the feature maps.
  (b) **ReLU:** Activation function applied after each batch normalization.
  (c) **Downsample:** Some bottleneck blocks include a downsampling component to match the dimensionality for the residual connection when the dimensions increase.

(3) **Additional Components in Bottlenecks:**
  - **Downsampling Sequential Block:** Specifically, in certain blocks where the depth or the number of channels increases, a downsampling layer is used in the shortcut path to match the output dimensions with the residual path. This typically includes a convolutional layer with a 1×1 kernel and a stride of 2, followed by batch normalization.

(4) **Final Layers:**
  - **AdaptiveAvgPool2d:** This layer performs adaptive average pooling to reduce each channel's spatial dimensions to a single scalar value, effectively summarizing the entire spatial information of the feature maps.
  - **Sequential (Fully Connected Layers):**
  (a) **Linear:** Converts the pooled features into a vector of size defined by the first linear layer (2048 to 256).
  (b) **ReLU and Dropout:** Activation and regularization are applied to prevent overfitting and to introduce non-linearity.
  (c) **Linear:** The final linear layer maps the features to the desired number of output classes (256 to 100 in this case).

This model structure leverages deep residual learning, enabling training of very deep network by using skip connections that help gradients flow through the network, thus mitigating the problem of vanishing gradients. Each component and configuration is carefully designed to optimize performance for fine-grained image analysis.

Model Specifications and Adjustments. The ResNet model setup includes several specifications and adjustments to optimize performance. Our team focused on two main aspects, respectively:

(1) A freezing layer strategy
(2) Inclusion of a final classifier layer

4.1.1 Freezing Strategy. Given our decision to utilize the weights from the pre-trained ResNet50 model, we needed to address the risk of catastrophic forgetting. To overcome this, our team elected to freeze the convolutional layers (*nn.Conv2d*) in the early layers and blocks. This approach helps retain the learned low-level features while allowing the later layers to adapt to the new tasks. By fine-tuning only the last layers, the model is updated with the pre-trained weights to achieve the specific objectives of the competition.

```
for name, child in model.named_children():
    if isinstance(child, nn.Conv2d):
        # Freeze this convolutional layer
        for param in child.parameters():
            param.requires_grad = False
    elif name in
    ['layer1', 'layer2', 'layer3', 'layer4']:
# These are the main blocks containing Conv2d layers
        for param in child.parameters():
            param.requires_grad = False
```

This choice is related to the competition condition of having a small dataset to train the model with and to accomplish the fine-grained image classification task.

4.1.2 Additional Layer - Final Classifier. Including an additional layer at the end of the model is crucial for dealing with feature extraction and dimensionality reduction. Considering that the early layers of a CNN learn fundamental features directly from the input image, and as we delve deeper, the layers begin to process more sophisticated and abstract information—sometimes unrelated to the specific task at hand—we decided to incorporate a fully connected layer. This layer enables more precise classification by fine-tuning the learned features to better match the task requirements. Additionally, it helps reduce the dimensionality of the feature maps and the number of parameters in the network, enhancing computational efficiency.

```
model.fc = nn.Sequential(
    nn.Linear(num_features, 256),
    nn.ReLU(inplace=True),
    nn.Dropout(0.1),
    nn.Linear(256, num_classes)
)

num_features = model.fc.in_features
```

All experiments on CNNs were conducted using identical optimizers, activation functions, and hyperparameters, as the primary intent was to evaluate different architectural modifications.

Observations:

- **ResNet with final classifier:** This setup showed configuration issues that need further investigation.
- **ResNet without freezing layers:** This approach led to retraining all layers, which was not ideal and showed signs of poor generalization.

4.2 Vision Transformers (ViT)

The Vision Transformers (ViT) adapts the principles of the Transformers architecture, initially designed for NLP, to the domain of computer vision. Unlike traditional models that typically rely on CNNs, ViT processes images through self-attention mechanisms, analyzing images as sequences of patches to capture complex inter-patch relationships.

Transformers revolutionized NLP by effectively handling sequences through self-attention, dynamically determining the relevance of different parts of the input data. ViT extends this concept to image analysis, treating images as sequences of flattened patches, similar to how words are handled in NLP. The model can be summarized through the following components:

- **Patch Embedding:** Images are divided into fixed-size patches (e.g., 16x16 pixels). Each patch is flattened and transformed into a high-dimensional vector through a linear projection. This step converts each patch into a token analogous to words in NLP models.
- **Positional Encoding:** To maintain spatial context, positional encodings are added to the patch embeddings. This allows the model to retain information about the positions of the patches within the image, essential for preserving the geometric structure of the image.
- **Transformers Encoder:** The sequence of embedded patches, enriched with positional information, is processed through multiple layers of the Transformers encoder. Each layer applies multi-headed self-attention and position-wise fully connected feed-forward networks. Layer normalization and residual connections are used to stabilize training and enhance gradient flow.
- **Classification Token:** An additional learnable classification token is appended to the sequence of patch embeddings. After passing through the Transformers layers, the state of this token is used to make the final classification.

Our implementation includes data management, model setup, training loop, and a sophisticated learning rate scheduling technique that combines a warmup phase with cosine annealing. Below is description of the code along with the theoretical background. Specifically, three implementations were employed. A comprehensive description of the commonalities among the various method implementations is given; subsequently, the specifications for the three models are provided. It is important to note that these are the three models effectively utilized during the competition (while the CNNs one was used only for experimenting).

Imports and Setup. After having included essential libraries for building and training NLP, image processing, data manipulation, plotting, and learning rate scheduling, a fixed random seed was set to ensure reproducibility. This helped in obtaining consistent results across different runs by fixing the randomness in data shuffling and weight initialization. Both parameters and hyperparameters were defined, including image dimensions, number of training epochs, batch size, and validation split ratio.

In the Dataset Management and Data Preparation classes, the dataset was loaded, extracted from the *.tar* or *.zip* format, and made available for the model. Additionally, data transformations for training and validation datasets was applied to the images, as described in sections 1.4 and 2.1, including resizing, normalization, and augmentation (only for the train part). Data augmentation is essential for improving the generalization of deep learning models. Our method includes random cropping, flipping and rotation. Additionally, normalization is applied to standardize the input data, ensuring that each feature has a mean of zero and a standard deviation of one. This helps in accelerating the convergence of the training process.

Model Setup and Training.

- Model Creation

Code 6 Model Creation

model_ft = timm.create_model('vit_large$_{p}atch16\_224', pretrained = True, num\_classes = Num\_class$)
    model = model.to(device)

As showed in the code, a pretrained Vision Transformers (ViT) model was loaded from the *timm* library and modifies the classification head to match the number of target classes. This includes the use of Multi-Head Attention, Scaled Dot-Product Attention and other architectural features seen in the Transformers architecture traditionally used for NLP. In particular, the timm module (also known as PyTorch Image Models) is an open-source library that provides an extensive collection of state-of-the-art image models for PyTorch. It includes pretrained weights and utility scripts for training, inference, and validation. To create our pretrained model, we used the create_model function from timm [8]. After the definition of the pretrained model, an optimizer (AdamW) and the learning rate scheduler with warmup and cosine annealing were defined as follows:

- Optimizer and Scheduler

Code 7 Optimizer and Scheduler

```
optimizer = optimizer_ft = torch.optim.AdamW(model_ft.parameters(),
lr=0.001, weight_decay=0.01)
num_warmup_steps = 500
num_training_steps = 15 * len(dataloaders['train'])
scheduler_ft = get_cosine_schedule_with_warmup(
optimizer_ft, num_warmup_steps =
num_warmup_steps, num_training_steps =
num_training_steps)
```

In particular: the **Warmup** gradually increases the learning rate from a small initial value to the maximum learning rate over the specified number of warmup steps and the **Cosine Annealing** decreases the learning rate following a cosine curve. This topic will be further amplified.

After the definition of those instruments, another important tool is specified:
Loss Function The model employs *LabelSmoothingCrossEntropy* as the criterion. This loss function helps prevent overfitting by smoothing the labels during training; for more details look section 2.8.
Then the model is defined and fitted:

Code 8 Defining and Fitting

```
model_ft = timm.create_model('vit_large_patch16_224', pretrained=True, num_classes=Num_class)
model_ft = model_ft.to(device)
model_ft = train_model(model_ft, criterion, optimizer_ft, scheduler_ft, num_epochs=num_epochs, checkpoint_path='checkpoints')
```

During the training, the loop iterates over epochs and batches of data. For each batch:

- Moves inputs and labels to the appropriate device (CPU or GPU);
- Performs a forward pass to compute the model outputs;
- Computes the loss;
- Performs a backward pass to compute gradients;
- Updates the model parameters using the optimizer;
- Adjusts the learning rate using the scheduler.

Learning Rate Scheduler with Warmup and Cosine Annealing. The learning rate scheduler is crucial for effective training. It combines two techniques: warmup and cosine annealing.

- Warmup Phase **Purpose:** Prevents large updates to model parameters at the start of training, ensuring stable and gradual learning.
  Implementation: Increases the learning rate linearly from a small initial value to the maximum learning rate over a predefined number of steps.
  Mathematical Formulation: The learning rate η_t at step t during the warmup phase is given by:
  $$\eta_t = \eta_{\text{init}} + \frac{t}{T_{\text{warmup}}}(\eta_{\text{max}} - \eta_{\text{init}})$$
  where:
  – η_{init} is the initial learning rate (typically close to zero);
  – η_{max} is the maximum learning rate at that point of the process;
  – T_{warmup} is the total number of warmup steps;
  – t is the current step.
- Cosine Annealing Phase **Purpose:** Efficiently fine-tunes the model parameters by gradually decreasing the learning rate in a cosine pattern, helping the model escape local minima and converge to a better solution.
  Implementation: After the warmup phase, the learning rate follows a cosine decay schedule. This means it starts at the maximum value and decreases following the cosine function.
  Mathematical Formulation: The learning rate η_t at step t during the cosine annealing phase (including the warmup phase) is given by:
  $$\eta_t = \eta_{\text{min}} + \frac{1}{2}(\eta_{\text{max}} - \eta_{\text{min}})\left(1 + \cos\left(\frac{t - T_{\text{warmup}}}{T_{\text{total}} - T_{\text{warmup}}}\pi\right)\right)$$
  where:
  – η_{min} is the minimum learning rate;
  – η_{max} is the maximum learning rate;
  – T_{total} is the total number of training steps;
  – T_{warmup} is the number of warmup steps;
  – t is the current step.

Therefore, these represent the common elements among the different methods used during the competition. Between the vit_fast and vit_slow, the primary difference lies in the adjustment of certain parameters, which rendered one model faster than the other. These include the batch size (12 and 32) and the learning rate (0.001 and 0.003). The code can be consulted here.

The most significant distinction between the third code and the first two is the difference in architecture:

- First and Second Model: They utilize a ViT model with a modified head (nn.Linear) for the desired number of classes. This model is simple and straightforward, focusing on the use of ViT for feature extraction.
- Third Model: It employs a combination of ViT with SpinalNet, adding additional fully connected layers with LeakyReLU. This approach aims to enhance feature learning by combining the power of ViT with the flexibility of SpinalNet.

4.2.1 *SpinalNet Architecture.*

The Vision Transformers model is enhanced with the SpinalNet architecture for the classification head. SpinalNet is designed to improve the deep learning model's performance by organizing neurons in a spinal structure, which helps in better gradient flow and generalization. The architecture splits the hidden layers into multiple segments, allowing gradients to flow more effectively during backpropagation. This approach reduces the chances of vanishing gradients, especially in very deep networks.

It should be noted that the activation function used is LeakyReLU, as it was found to be the best for our model in terms of accuracy following experiments. An additional difference is the presence of the following two elements in the third model compared to the first two:

Code 9 SpinalNet Architecture

```
class SpinalNet(nn.Module):
def __init__(self)
def forward(self, x)
model_ft.head = SpinalNet()
```

- **Early Stopping**: All our models implemented an early stopping mechanism to halt training when the model ceased to reduce its validation loss.
- **Label Smoothing Cross-Entropy**: It introduces cross-entropy loss with label smoothing, enhancing the model's robustness and improving generalization. The first two models simply use the standard cross-entropy loss.

Additionally, all the models operate with checkpoints that save the results and allow tracking of intermediate results. This enables the training process to resume from the last checkpoint in case of an interruption. Finally, all models conclude with a submission function that calculates the predictions and returns an accuracy score.

Preparing and Submitting Predictions.

- **Preparing the Submission Dictionary**: The collected predictions are stored in *sub*. A new dictionary *sub1* is initialized with a key for images and the group name (*'Tanos Matadores'*).
- **Populating the Dictionary**: For each item in *sub*, the function extracts the class name and stores it in the *sub1['images']* dictionary.
- **Submitting the Results**: Finally, the *sub1* dictionary is passed to the *submit* function to submit the results to the competition server.

4.2.2 *Competitiveness and Advantages.*

ViT offers several advantages that make it highly competitive in the field of computer vision:

- **Scalability:** The Transformers architecture allows ViT to scale effectively with increases in model size, data volume, and computational resources, showing improved performance when trained on large-scale image datasets.
- **Global Reception Field:** The self-attention mechanism enables the model to consider global dependencies between any parts of the input image. This is in contrast to CNNs, which typically require deeper architectures and pooling layers to increase their receptive fields.

The Vision Transformers represents a significant shift in how deep learning models can be applied to computer vision tasks. By leveraging the power of self-attention and Transformers-based architectures, ViT achieves exceptional performance on standard benchmarks, challenging the dominance of CNNs in large-scale image recognition tasks.

5 EXPERIMENTS

In this section, we present the experiments conducted to develop our model. These experiments encompass various datasets, including Oxford Flowers 102, FGVC Aircraft, and Stanford Cars. Here, we report the most significant experiments that led to the final model described in the Methods section. In total, we report 43 experiments, from which we keep the loggings in notes that could be shared with the readers at request.

5.1 Flowers Dataset: Phase 1 Experiments

For this experiment, the Flower102 dataset was utilized, a collection specifically designed for fine-grained image classification tasks, focusing on distinguishing among different types of flowers. This dataset includes 8,189 images across 102 distinct classes, representing a broad variety of common flower species from the United Kingdom. Each class corresponds to a single species, such as "Bluebell" or "Snowdrop". The dataset is organized into 6,149 training images and 2,040 test images, providing a balanced distribution to support both the training and validation of machine learning models.

Each flower class within the dataset is precisely defined, capturing the unique floral characteristics necessary for accurate classification. Annotations in the dataset also include segmentation masks that outline the shape of each flower, aiding in tasks that require precise object segmentation and recognition.

Table 1, as mentioned in the document, presents the results of our experiments tested on the Flower102 dataset, showcasing the effectiveness of each configuration in terms of accuracy. This approach aims to further the understanding of how different architectural tweaks can influence performance in fine-grained image classification tasks.

Table 1: Experiments on flower102 dataset

Experiment	Comments	Test accuracy
1	ResNet 152: Basic model implementation	56.32%
2	ResNet50: Reduced model complexity, higher overfitting observed	34.70%
3	ViT Large Patch16_224 without Background Classes: Improved handling of relevant features	93.43%
4	ViT Large Patch16_224 with Background Classes: Further improvement in accuracy by considering background noise	98.25%

5.2 Aircrafts Dataset: Phase 2 Experiments

For our experiments, the FGVCAircraft dataset from PyTorch's torchvision package was employed, known for its applicability in fine-grained visual categorization of aircraft. The dataset comprises over 10,000 images across 100 different aircraft model categories, such as "Boeing 737", "Airbus A320", and "Lockheed Martin F-22 Raptor". Each category specifically targets a distinct model, providing an excellent resource for machine learning tasks that demand high precision in image classification. These specific labels help to ensure that the models trained on this dataset can distinguish finely between different types of aircraft, a crucial capability for applications that require precise aircraft identification.

Experiment	Comments	Test accuracy
1	Code contained data leakage between sets, corrected in next experiment	84.21%
2	Model rapidly overfits reaching 97.5% of train accuracy and 74.8% of val accuracy in the first epoch. Run stopped at the 4th epoch.	76.96%
3	We added more aggressive augmentation through many v2.transforms functions. 10 epochs finished after 23 minutes, the 46.7% in val accuracy suggests we need more epochs.	52.96%
4	Increasing epochs from 10 to 25. Run stops at epoch 14 because of Azure VM failure (later we'll learn about checkpoints). We (wrongly) deduced LR was too small.	73.32%
5	We replaced the optimizer: AdamW with an LR=0.01 instead of the previous SGD. By epoch 3, we begin seeing "nan" in the losses. We thought the optimizer was wrongly implemented, but now we realize gradients exploded due to the high LR. AdamW was discarded for the time being (mistake).	0.99%
6	Re-run experiment 2 for quality check and we realize results are different now. This motivates using a torch seed moving on.	72.33%
7	We added the seed and even more augmentation. Azure VM turns off after 3rd epoch.	-
8	We added checkpoints and Early Stopping to the training function. The run retrieved minimal accuracy values, so we deduced our functions were incorrectly implemented.	-
9	Back to experiment 2 structure, keeping the Early Stopping, the last augmentations, and 15 epochs instead of 25. In retrospect, it was an error not to insist on the checkpoints here.	71.41%
10	Switch from v2.transforms augmentations to automatic ones (CutMix, CutOut, MixUp, RandAugment). 10 epochs. Optimism with the final results, the 51.9% train accuracy suggests that a little less regularization would help achieve a healthy bias-variance trade-off.	73.00%
11	Back to 15 epochs. We reduced to half the alphas of each of the automatic augmentations and added LabelSmoothing. Seems like we are close to a balance.	73.90%
12	Added checkpoints again, increased epochs to 25, took all augmentation away, and changed the LR Scheduler from StepLR to ReduceLROnPlateau with patience=3. Best run so far. Yet 99.2% of training accuracy suggested regularization was needed.	79.36%
13	We bring back the v2.transforms augmentations to regularize the model. Again we achieved our best run so far, but the 99.3% train accuracy suggested more regularization was necessary.	82.51%
14	We doubled the LR to check how stable would be a faster model. We reach 97% train accuracy at epoch 14, but we think with more regularization we could have a good and fast model.	80.17%
15	We brought back MixUp-CutMix-RandomAugment. Train accuracy peaked at 51%. We realized combining both augmentations was too much.	78.06%
16	With Exp. 14 as a baseline, we added weight_decay=1e-4 to the SGD optimizer. After epoch 9 we reached a local minima and we barely improved from there.	79.21%
17	To escape the local minima we changed the LR Scheduler for PyTorch's CosineAnnealingWarmRestarts.	80.59%
18	Hyperparameter tuning trial 1: Grid search over the following values finishes when VM turns off after an hour. Values: 'weight_decay': [1e-5, 1e-4, 1e-3], 'learning_rate': [3e-3, 5e-4, 1e-4]	-
19	Hyperparameter tuning trial 2: Same as before but with a random search algorithm instead of the grid. Again, the VM turned off and ruined the experiment. We repeated the experiment three times decreasing the possible hyperparameters, none worked and we abandoned the idea of automatic hyperparameter tuning.	-
20	Batch size changes from 12 to 32 and Label Smoothing p=0.3 (before it was 0.1). p=0.3 seems like an overshooting, we'll try with 0.2 in the next run. In retrospect, we should have considered the effects of changing batch size separately.	78.21%
21	Label Smoothing with p=0.2. Generalization error looks promising. We'll keep this model as a baseline moving on.	81.09%
22	After reading the paper on the Bag of Tricks, we copied their LR=0.01 keeping the rest from Exp. 21. We got trapped in a local minimum and realized a 10x increase in LR was excessive.	73.51%
23	We test a warmup to the LR Scheduler, on just 15 epochs. Results look promising (just 49% of train accuracy) so we'll keep this LR scheduler moving on	66.06%
24	Back to Exp. 21 as a baseline. Changed LR Scheduler from ReduceLROnPlateau to transformers' cosine scheduler with warmup. Initialization fails and we disconnect the model after 5 epochs.	0.99%
25	Went back to Label Smoothing at 0.1 and added CutMix (alpha=0.5) & MixUp (alpha=0.1), the rest of the augmentation was the basic geometric one. Regularization is too high and we barely reach 32% train accuracy at epoch 21.	63.13%
26	We reduce the alphas from previous run to the half and take Label Smoothing out. We rule out the use of CutMix-MixUp-RandAugment for being too strong regularizators.	65.89%
27	Changed activation function, from ReLU to LeakyReLU	79.65%

Table 2: Experiments over the FGVC Aircrafts dataset

5.3 Cars Dataset: Phase 3 Experiments

For this experiment, the *Stanford Cars dataset* was used [1]. It is a collection specifically designed for fine-grained image classification, focusing on distinguishing among different makes, models, and years of cars. This dataset includes 16,185 images, encompassing 196 distinct classes of cars. The images are neatly divided into two subsets: 8,144 training images and 8,041 testing images, ensuring a near-even split to facilitate both training and validation of machine learning models.

Each class within the dataset represents a specific type of car, detailed down to the make, model, and year, such as a "2012 Tesla Model S" or a "2012 BMW M3 coupe". The annotations include bounding boxes that specify the location of the car within each image, which is particularly useful for tasks involving object detection and localization.

Compared to the tests conducted on the Aircrafts dataset, a warm-up cosine annealing learning rate scheduler (see the Related Works section) and an AdamW optimizer were introduced. Under these conditions, experiments were conducted on different activation functions (also described in the Related Works section). Table 3 shows the results, in terms of accuracy, of our experiments varying the activation function used.

Table 3: Experiments on Cars Dataset

Experiment	Description	Val. accuracy
1	Activation function: ReLU	87.0%
2	Activation function: LeakyReLU	88.5%
3	Activation function: PReLU	69.9%
4	Activation function: Swish	75.9%
5	Activation function: GELU	69.2%
6	Activation function: Sigmoid	69.0%
7	Attempted to replicate the best model from Aircrafts trials onto the Cars dataset. Used the same augmentation strategies.	85.33%
8	Reduced weight decay from 0.01 to 0.001 to test effects on overfitting, based on the last baseline model. Stopped early at epoch 25 due to unsatisfactory results.	72.44%
9	Increased the batch size from 32 to 50. Quick epochs, less than 4 minutes each. Suboptimal performance.	71.70%
10	Tripled the learning rate based on the last baseline model VIT_CARS_87. Achieved significant improvement in validation accuracy.	94.60%
11	Reduced LR at half (0.001 to 0.0005)	95.46%
12	ViT_fast with the mean and standard deviation function applied. Training stopped due to local minima.	75.57%

Therefore, the most suitable activation function for our dataset is LeakyReLU. We record this result and proceed with our experiments on the same dataset, albeit with certain modifications.

6 RESULTS

The results of our classification emerged from applying our models to the test images present in the dataset. Specifically, the models, trained on the training images, were then applied to the test set to calculate the accuracy of our predictions. Subsequently, the predictions were submitted through a submit function described in the Methods section, and the accuracy was calculated by comparing our predictions with the actual class labels of the images.

During the competition, only the models *vit_fast* and *vit_WITH_spiralnet* produced predictions that were tested and returned accuracies; the *vit_slow* model, on the other hand, was trained during the competition but was not used for predictions. However, as previously stated, the main difference between the two methods was primarily the batch size and learning rate, while the model's operational structure was almost identical.

The accuracies provided by the models *vit_fast* and *vit_WITH_spiralnet* were 72% and 71.8%, respectively, making the models competitive. Checkpoints were saved for all the models, which include intermediate results showing the accuracy on the training set and the validation set epoch by epoch.

From the results presented in tables 5, 4 and 6, it is possible to observe that our models were not overfitting, as the accuracy on the training images was lower than the accuracy on the validation images. The only case in which the training accuracy exceeded the validation accuracy is that of the *vit_fast* model, which nonetheless represents a controlled case. Additionally, we cannot speak of underfitting given the slight disparity between the two accuracies.

Epoch	Train Accuracy	Validation Accuracy
0	0.0014	0.0006
1	0.1999	0.5807
2	0.6816	0.7747
3	0.7588	0.8066
4	0.7774	0.8048
5	0.7867	0.8152
6	0.7925	0.8226
7	0.7958	0.8287
8	0.7963	0.8367
9	0.7972	0.8392
10	0.7967	0.8386
11	0.7976	0.8336
12	0.7974	0.8386
14	0.7980	0.8428
15	0.7986	0.8478
16	0.7981	0.8478
17	0.7987	0.8521
18	0.7980	0.8386
19	0.7983	0.8484
22	0.8000	0.8521
23	0.8000	0.8508

Table 4: Training and validation accuracies for the vit_slow model. Submit: accuracy 0.7200

As can be observed, the *vit_slow* model exhibited the best accuracy on the validation set. However, issues related to the match

Epoch	Train Accuracy	Validation Accuracy
0	0.0040	0.0050
1	0.3370	0.6410
2	0.6832	0.7100
3	0.7264	0.7150
4	0.7494	0.7260
5	0.7654	0.7340
6	0.7814	0.7280
7	0.7908	0.7280
8	0.7958	0.7300
9	0.7970	0.7430
10	0.7986	0.7290
11	0.7992	0.7480
12	0.7992	0.7320
13	0.7988	0.7430
14	0.7996	0.7400
15	0.7996	0.7460
16	0.7998	0.7360
17	0.8000	0.7540
18	0.8010	0.7570
19	0.8100	0.7540

Table 5: Training and validation accuracies for the vit_fast model. Submit: not submitted but trained

between the submission function and the dataset structure during the reading phase prevented the delivery of the accuracy associated with this model. Additionally, note that the *vit_WITH_spiralnet* model has 5 extra epochs. This is due to a second run of the code; after proceeding with 20 epochs, the training was extended to 25 epochs to check if the result had stabilized.

To summarize, the best accuracy achieved on the test set was 72%, classifying our model as capable of capturing the variety of images reasonably well, although not perfectly.

7 CONCLUSION

In conclusion, this report aims to primarily explore two methods for image classification CNNs and ViT, focusing mainly on the ViT, which represents the structure effectively utilized during the competition. Utilizing the provided dataset, referencing images of various mammal species, two of the tested models (of ViT type) led to an accuracy of approximately 72%. Among the primary challenges of the project are, first and foremost, the search for models for the FGIC task, followed by the search for suitable datasets to train the models and conduct experiments. Additionally, the dataset size, understanding how to handle datasets of various formats, and how to make them accessible to our models in the most possible general way. This is compounded by the understanding of which techniques to employ for data cleaning and data augmentation, as well as the choice of parameters to use. Tools such as activation functions, learning rate schedulers, spiralnet structure, and the optimizer have also been subjects of research and challenges. Regarding the limitations of this study, experiments were primarily conducted on three types of datasets (flowers, aircraft, and cars), which allowed us to focus on the choice of parameters and the aforementioned functions/structures. For example, not all existing activation functions were utilized, nor were all possible values of the learning rate or different optimizers. This is also reflected in the choice of models; after careful evaluation, ViT proved to be our best method, but this does not preclude the existence of structures better suited to support this task. Overall, the work has been satisfactory and has allowed for the conduct of numerous experiments by manipulating various parameters and understanding their intrinsic significance.

Epoch	Train Accuracy	Validation Accuracy
0	0.0064	0.0020
1	0.0180	0.2160
2	0.1018	0.4640
3	0.2518	0.5620
4	0.3768	0.5870
5	0.4546	0.6540
6	0.5114	0.6570
7	0.5486	0.6860
8	0.5772	0.6950
9	0.6000	0.6960
10	0.6188	0.7180
11	0.6394	0.7260
12	0.6504	0.7140
13	0.6712	0.7240
14	0.6848	0.7360
15	0.6920	0.7380
16	0.7060	0.7310
17	0.7156	0.7360
18	0.7234	0.7400
19	0.7142	0.7380
20	0.7564	0.7350
21	0.7388	0.7500
22	0.7494	0.7570
23	0.7788	0.7420
24	0.7758	0.7440
25	0.7792	0.7480

Table 6: Training and validation accuracies for the *vit_WITH_spiralnet* model. Submit: accuracy 0.7180

REFERENCES

[1] 2013. Stanford Cars Dataset. https://www.kaggle.com/datasets/jessicali9530/stanford-cars-dataset

[2] 2020. Image Recognition at Scale. https://arxiv.org/pdf/2010.11929

[3] 2021. Fine-Grained Image Analysis with Deep Learning: A Survey. https://www.pure.ed.ac.uk/ws/portalfiles/portal/248228355/Fine_Grained_Image_Analysis_WEI_DOA13112021_AFV.pdf

[4] 2021. How to Train Your ViT? Data, Augmentation, and Regularization in Vision Transformers. https://arxiv.org/pdf/2106.10270

[5] 2023. Albumentations Documentation. https://albumentations.ai

[6] 2023. Fine-Grained Image Classification. https://paperswithcode.com/task/fine-grained-image-classification

[7] 2023. LeakyReLU. https://pytorch.org/docs/stable/generated/torch.nn.LeakyReLU.html

[8] 2023. PyTorch Image Models (timm). https://github.com/huggingface/pytorch-image-models

[9] Amir Ahmad and Shehroz S. Khan. 2019. Survey of State-of-the-Art Mixed Data Clustering Algorithms. *IEEE Access* 7 (2019), 31883–31902. doi:10.1109/ACCESS.2019.2903568

[10] Christian Bauckhage. 2015. NumPy/SciPy Recipes for Data Science: K-Medoids Clustering. *ResearchGate* (2015).

[11] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven E. Whang, and Jennifer Widom. 2009. Swoosh: A Generic Approach to Entity Resolution. *The VLDB Journal* 18, 1 (2009), 255–276.

[12] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2019. End-to-End Entity Resolution for Big Data: A Survey. *arXiv preprint arXiv:1905.06397* (2019).

[13] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2020. An Overview of End-to-End Entity Resolution for Big Data. *Comput. Surveys* 53, 4 (2020).

[14] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic Data Cleaning: Putting Violations into Context. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 458–469.

[15] Halbert L. Dunn. 1946. Record Linkage. *American Journal of Public Health* 36, 12 (1946), 1412–1416.

[16] Jérôme Euzenat and Pavel Shvaiko. 2013. *Ontology Matching*. Springer.

[17] Ivan P. Fellegi and Alan B. Sunter. 1969. A Theory for Record Linkage. *J. Amer. Statist. Assoc.* 64, 328 (1969), 1183–1210.

[18] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. 2014. *Mining of Massive Datasets*. Cambridge University Press.

[19] Yiming Lin, Hongzhi Wang, Jianzhong Li, and Hong Gao. 2016. Efficient Entity Resolution on Heterogeneous Records. *arXiv preprint arXiv:1610.09500* (2016).

[20] Matthew Michelson and Craig A. Knoblock. 2006. Learning Blocking Schemes for Record Linkage. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 440–445.

[21] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2020. Blocking and Filtering Techniques for Entity Resolution: A Survey. *Comput. Surveys* 53, 2 (2020), 31:1–31:42. doi:10.1145/3377455

[22] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. 2017. Searching for Activation Functions. *arXiv preprint arXiv:1710.05941* (2017).

[23] Peter J. Rousseeuw. 1987. Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis. *J. Comput. Appl. Math.* 20 (1987), 53–65.

[24] Robert Tibshirani, Guenther Walther, and Trevor Hastie. 2001. Estimating the Number of Clusters in a Data Set via the Gap Statistic. *Journal of the Royal Statistical Society: Series B* 63, 2 (2001), 411–423.

[25] Stijn M. Van Dongen. 2000. *Graph Clustering by Flow Simulation*. Ph. D. Dissertation. Utrecht University.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *Advances in Neural Information Processing Systems* (2017).

[27] Xiu-Shen Wei, Yi-Zhe Song, Oisin Mac Aodha, Jianxin Wu, Yuxin Peng, Jinhui Tang, Jian Yang, and Serge Belongie. 2022. Fine-Grained Image Analysis with Deep Learning: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 12 (2022), 8927–8948. doi:10.1109/TPAMI.2021.3126648

[28] Xiang Zhang, Qingqing Yang, Jinru Ding, and Ziyue Wang. 2020. Entity Profiling in Knowledge Graphs. *arXiv preprint arXiv:2003.00172* (2020).