CS301 Introduction to AI

Final Project Report

Vertically Flipped Image Detection

Anh, Dinh Nguyet - 2059003

Content

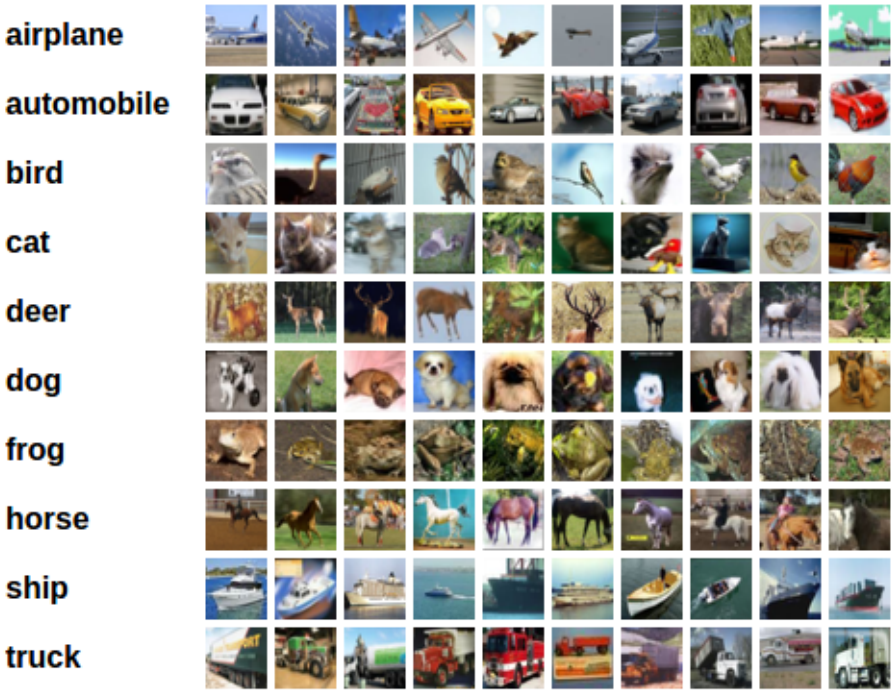
1. Dataset Preparation

1.1 Dataset Overview

The CIFAR-10 dataset was developed along with the CIFAR-100 dataset by researchers at the CIFAR institute [1]. The dataset comprises 60,000 32×32 pixel color photographs of objects from 10 classes, such as frogs, birds, cats, ships, etc. The class labels and their standard associated integer values are listed below.

- 0: airplane
- 1: automobile
- 2: bird
- 3: cat
- 4: deer
- 5: dog
- 6: frog
- 7: horse
- 8: ship
- 9: truck

These are **very small images**, much smaller than a typical photograph, and the dataset was intended for computer vision research. It is also widely used for benchmarking computer vision algorithms, relatively straightforward to achieve 80% classification accuracy [2].



1.2 Dataset Selection

Comparison with other recommended choice, i.e. STL-10

- Image Complexity: CIFAR-10 images are 32x32 color images, meaning more complexity than others, which are 28x28 grayscale images. So, it offers a more realistic benchmark for image classification tasks.
- Variety and Balance: STL-10 has 500 training images (10 pre-defined folds), 800 test images per class [3], which is too small for supervised learning models in general, fewer labeled training examples (5,000 labeled images vs. 50,000 in CIFAR-10)
- Relevant: For a task focused on detecting vertical flips in images, the diversity and real-world relevance of images in CIFAR-10 (spanning animals, vehicles, and other categories) make it an appropriate choice.
- Educational Value: While STL-10, MNIST, and Fashion-MNIST are excellent for specific purposes (e.g., STL-10 for semi-supervised learning, MNIST for handwriting recognition), CIFAR-10 has a balance between complexity and accessibility.

Finally, I decided to choose CIFAR-10 as dataset for this project:

- Ease of access: CIFAR-10 is **readily accessible through PyTorch's** dataset utilities, allowing straightforward integration using `torchvision`, efficient batching and parallel data loading with PyTorch's `DataLoader`

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)
```

Image source

- Augments data on-the-fly with random vertical flips during the data loading process, offering more variability across epochs.
  The `vflip()` transform performs a vertical flip of an image, with a given probability.

1.3 Dataset Splitting

The CIFAR-10 dataset, readily accessible through PyTorch, has a standard division of 50,000 training images and 10,000 test images. This pre-established split is important for training and evaluating machine learning models. Therefore, I decided to utilize the readily available divided datasets on PyTorch as mentioned above.

1.4 Dataset Loading

This is the first, and one of the most important steps as it includes data normalization: transform the set of data to be on a similar scale. Normalizing data to a range of -1 to 1 or 0 to 1, by subtracting the mean and dividing by the standard deviation, is essential in machine learning to standardize and improve model training [4].

As Baran Karakus explains on Stack Overflow, "If you want to compute means on an image-by-image and channel-by-channel basis, use axis=(1,2). If you want to compute means over all of your images per channel, use axis=(0,1,2)" (Karakus, 2020) [5].

My initial approach to calculate the mean and standard deviation for better model training later on:

```
# cifar10
train_set = datasets.CIFAR10(root='../data/', train=True, download=True, transform=train_transform)
print(train_set.train_data.shape)
print(train_set.train_data.mean(axis=(0,1,2))/255)
print(train_set.train_data.std(axis=(0,1,2))/255)
# (50000, 32, 32, 3)
# [0.49139968  0.48215841  0.44653091]
# [0.24703223  0.24348513  0.26158784]
```

However, the dimensionality of the data tensor doesn't match the expectations when calculating the mean and standard deviation as the tensor does not have the shape I assumed when stacking.

```
import torch
import torchvision
import torchvision.transforms as transforms

# Load the CIFAR-10 training dataset with ToTensor() transformation
train_set = torchvision.datasets.CIFAR10(root='./data',
                                         train=True,
                                         download=True,
                                         transform=transforms.ToTensor())

# Stacking and reshaping
loader = torch.utils.data.DataLoader(train_set,
                                     batch_size=len(train_set),
                                     shuffle=False, num_workers=0)

# Extract the entire dataset from the DataLoader
data, _ = next(iter(loader))

# calculate mean, and sd
mean = data.mean(dim=[0, 2, 3])
std = data.std(dim=[0, 2, 3])

print(f"Mean: {mean}")
print(f"Std: {std}")
```

```
Files already downloaded and verified
Mean: tensor([0.4914, 0.4822, 0.4465])
Std: tensor([0.2470, 0.2435, 0.2616])
```

My solution is to output the Image Data Structure `(N, C, H, W)` with the help of PyTorch's `DataLoader` to ensure the correct shape where:

- N: number of images; `batch_size`,
- C: number of channels (3 for RGB),
- H: the height of the images,
- W: the width of the images

Each image has dimensions corresponding to [C, H, W]

The dimensions [0, 2, 3] correspond to the batch, height, and width dimensions.

From the result, I can now normalize the dataset and proceed to the next step, which is to plug them in the normalization method, loading CIFAR-10 datasets with updated transform.

```
transform = transforms.Compose([
   transforms.ToTensor(),
   transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
])
```

1.5 Data Labeling

```
class BalancedFlipCIFAR10(Dataset):
    def __init__(self, cifar_dataset):
        self.cifar_dataset = cifar_dataset
        self.flip_labels = torch.randint(0, 2, (len(cifar_dataset),))

    def __len__(self):
        return len(self.cifar_dataset)

    def __getitem__(self, idx):
        image, original_label = self.cifar_dataset[idx]
        flip_label = self.flip_labels[idx]
        if flip_label == 1:
            image = transforms.functional.vflip(image)
        return image, flip_label, original_label
```

Initiation

```
def __init__(self, cifar_dataset):
    self.cifar_dataset = cifar_dataset
    self.flip_labels = torch.randint(0, 2, (len(cifar_dataset),))
```

Class `BalancedFlipCIFAR10` inherits from Dataset, a PyTorch class representing a dataset. The `__init__` method is the constructor, which is called when a new instance of BalancedFlipCIFAR10 is created.

Then, it would store the passed `cifar_dataset` object (instance) within the dataset object:

```
self.cifar_dataset = cifar_dataset
```

loaded via `torchvision.datasets.CIFAR10` *(contains the images and labels from the CIFAR-10 dataset, later be used to access images)*

Next, it generates a tensor of random integers (0 or 1) by:

```
self.flip_labels = torch.randint(0, 2, (len(cifar_dataset),))
```

Each image in the dataset should be flipped vertically (1) or kept as is (0).

The use of `torch.randint(0, 2, (len(cifar_dataset),))` ensures that each image has an equal chance of being assigned a flip label of 0 or 1.

Data augmentation

```
def __getitem__(self, idx):
    image, original_label = self.cifar_dataset[idx]
    flip_label = self.flip_labels[idx]
    if flip_label == 1:
        image = transforms.functional.vflip(image)
    return image, flip_label, original_label
```

It retrieves an image from the `cifar_dataset` based on the provided index `idx`. The CIFAR-10 dataset is composed of images and their corresponding labels, this method only uses both the original label and the binary label of the images

```
image, original_label = self.cifar_dataset[idx]
```

It then looks up the `flip_label` for this particular image. A `flip_label` of 1 means this image should be flipped vertically; 0 means it should remain unchanged.

If `flip_label` is 1, the image is flipped vertically using:

```
image = transforms.functional.vflip(image)
```

Finally, it returns the potentially flipped image along with its `flip_label` to the caller. This means each item retrieved by this method is a tuple of an image tensor and a binary label showing if the image has been flipped. I then apply the BalancedFlipCIFAR10 class as followed

```
balanced_train_dataset = BalancedFlipCIFAR10(trainset)
balanced_test_dataset = BalancedFlipCIFAR10(testset)
```

2. Model Selection

2.1 ResNet18

2.1.1 Overview

ResNet, or Residual Network, introduces a groundbreaking deep residual learning framework, significantly aiding the training of much deeper neural networks. This architecture is pivotal for its innovative use of residual blocks, allowing information to bypass one or more layers through shortcut connections. This design addresses the vanishing gradient problem, facilitating the training of networks with a large number of layers [6].

Among the various ResNet configurations, ResNet18 is chosen for its optimal balance between depth and computational efficiency. This balance makes it particularly suitable for tasks like classifying CIFAR-10 images, where both accuracy and performance are key considerations [7]. ResNet18's architecture, while deep enough to learn complex patterns, remains computationally manageable for moderate hardware capabilities.

2.1.2 Model setup

To set up ResNet18 for a binary classification task, the network's final fully connected layer is modified to output a single value. This adjustment allows the model to distinguish between two classes, in this case, whether images from the CIFAR-10 dataset are flipped or not.

Utilizing ResNet18_Weights.IMAGENET1K_V1 ensures the model benefits from pretraining on the extensive ImageNet dataset, providing a solid foundation of learned features.

```
from torchvision.models import resnet18, ResNet18_Weights

model_resnet18 = resnet18(weights=ResNet18_Weights.IMAGENET1K_V1)
num_ftrs = model_resnet18.fc.in_features
model_resnet18.fc = nn.Linear(num_ftrs, 1)
```

- `ResNet18_Weights.IMAGENET1K_V1` specifies that it's loading the ResNet-18 model pretrained on the ImageNet dataset.

acc@1 (on ImageNet-1K)	69.758
acc@5 (on ImageNet-1K)	89.078

Image source

- By specifying `weights=ResNet18_Weights.IMAGENET1K_V1`, I explicitly choose the set of weights, which makes the code more transparent and future-proof.

2.2 VGG16

2.2.1 Overview

VGG16 is a deep convolutional neural network model utilized for image classification tasks.

The model comprises 16 layers of artificial neurons, each incrementally processing image information to enhance prediction accuracy. VGG16 employs convolution layers with a 3x3 filter and stride 1, maintaining the same padding, and utilizes a maxpool layer with a 2x2 filter of stride 2. [8] This configuration of convolution and max pool layers is consistently applied throughout the architecture. Ultimately, the network includes two fully connected layers, culminating in a softmax function for output.

VGG16, pre-trained on the extensive ImageNet database, classifies images into 1,000 categories with a 92.7% top-5 accuracy, effectively identifying various objects, animals, and plants in new images. [8]

I chose VGG16 for its simplicity and effectiveness in feature extraction, however, there were few adjustments made to simplify the process.

2.2.2 Model setup

Same as ResNet18, I'm utilizing Weights.IMAGENET1K_V1, which can be imported directly from PyTorch, to ensure the model benefits from pretraining on the extensive ImageNet dataset, providing a solid foundation of learned features.

First, we need to load the VGG16 model pre-trained on ImageNet and modify its classifier to suit binary classification tasks (flipped vs. not flipped).

```
from torchvision.models import vgg16, VGG16_Weights
model_vgg16 = vgg16(weights=VGG16_Weights.IMAGENET1K_V1)
```

Secondly, I need to extract features from images using `model_vgg16.features`, which contains the convolutional layers of the VGG16 model.

```
for param in model_vgg16.features.parameters():
   param.requires_grad = False
```

Then, I iterated over all parameters (weights and biases) in these layers and set `requires_grad` to `False`. In other words, I am freezing these parameters, that they will not be updated (or learned) during the training process. This action will make the fine-tuning process more efficient while preserving learned features, and preventing overfitting [9].

The VGG16 model ends with a classifier section, typically used to classify the extracted features into 1000 ImageNet classes [10]. Therefore, to create a new layer that matches the input size of the existing final layer, I retrieved the number of input features to the final layer in the classifier by:

```
num_features = model_vgg16.classifier[6].in_features
```

Then, I replace the last layer of the classifier with a new linear layer

```
model_vgg16.classifier[6] = nn.Linear(num_features, 1)
```

The single output will then be used, with a sigmoid activation function, to *predict the probability of an image being in one of the two classes*.

3. Training Process

3.1 Training setup

The training approach performs efficient training of different neural network architectures for binary classification tasks. The `train_model` function, designed with flexibility and reusability in mind, allows it to be used for training diverse models, including ResNet18 and VGG16.

3.2 Function Design

```python
def train_model(model, criterion, optimizer, train_dataset, num_epochs=10):
    model.train()
    train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
    for epoch in range(num_epochs):
        running_loss = 0.0
        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels.unsqueeze(1).float())
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f'Epoch {epoch + 1}, Loss: {running_loss / len(train_dataset)}')
    print('Finished Training')
```

The `train_model` function accepts a `model`, loss function `criterion`, `optimizer`, `train_dataset`, and the `number of epochs` as inputs. This ensures that the function can adapt to various model architectures without modification.

1. The `criterion: BCEWithLogitsLoss()` with `BCE` stands for Binary Cross-Entropy[1] is a Loss Function that will measure how far our model's predictions are from true labels. This is also used by Logistic Regression [11]. The term "logits" refers to the raw output of the last layer of a neural network before applying the sigmoid activation function[2]. Combining the sigmoid activation and the binary cross-entropy loss into one step (BCEWithLogitsLoss) provides **numerical stability advantages** over applying them separately.
2. The `optimizer`: The `SGD` optimizer[3] adjusts the weights of the network to minimize the loss computed by `criterion`. The use of optimizer at this point is to save time and resources.
3. The `learning rate` controls how much to change the model in response to the estimated error each time the model weights are updated. In this case, `lr=0.001`
4. `Momentum` helps accelerate SGD in the relevant direction. It does this by adding a fraction (`momentum=0.9` in this case) of the update vector of the past time step to the current update vector.

To put it lightly, the loss function calculates how well the model is performing, and the optimizer adjusts the model parameters to improve performance.

---

[1] Cross-Entropy measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-Entropy loss increases as the predicted probability diverges from the actual label, making it an effective measure for classification. [12]

[2] Sigmoid activation function applies some non-linear function after each matrix multiplication so the network can learn any necessary functional relationship and not just linear ones. [13]

[3] SGD calculates the gradient value over each example of the training dataset and updates the parameters(weights) accordingly [14]

3.3 Usage in training models

For the project at hand, the `train_model` function has been instrumental in training both ResNet18 and VGG16 models, adapted for binary classification tasks:

- ResNet18: Leveraging its depth and residual connections, ResNet18 is adapted to **predict a binary outcome**. The function facilitates its training by efficiently managing the forward and backward passes, loss computation, and parameters' optimization over several epochs.
- VGG16: Similarly, VGG16, known for its simplicity and deep architecture, is fine-tuned for the same binary classification task. Despite the architectural differences from ResNet18, the `train_model` function seamlessly applies, underscoring the function's adaptability.

3.4 Challenges and Considerations

3.4.1 Consistency Across Models

A crucial aspect of employing this function across different models is ensuring consistency in how models are adapted for binary classification, particularly in modifying the final layer to output a single value.

3.4.2 Hyperparameters Tuning

While the function supports diverse architectures, optimal training outcomes for each model may require specific hyperparameters adjustments, such as learning rate or batch size, which are external to the function's scope. *See Section 4.2.*

4. Testing and Evaluation

4.1 Model Performance

4.1.1 ResNet18

According to the result, we could say that the model performs relatively well and balanced on the binary classification task of predicting whether images from the CIFAR-10 dataset are flipped or not flipped.

```
Epoch 1, Loss: 0.00752139183908701
Epoch 2, Loss: 0.004953617543578148
Epoch 3, Loss: 0.0038913180029392243
Epoch 4, Loss: 0.0030067858758568763
Epoch 5, Loss: 0.0022742105039954183
Epoch 6, Loss: 0.0017798942447826266
Epoch 7, Loss: 0.00142927748369053
Epoch 8, Loss: 0.001111596397217363
Epoch 9, Loss: 0.0009349017628934235
Epoch 10, Loss: 0.0008202343305340037
Finished Training
```

```
model_resnet18.load_state_dict(torch.load(PATH))
model_resnet18.eval()

correct = 0
total = 0
test_loader = DataLoader(balanced_test_dataset, batch_size=64, shuffle=False)
with torch.no_grad():
    for images, flip_labels, original_labels in test_loader:
        outputs = model_resnet18(images)
        predicted = torch.round(torch.sigmoid(outputs)).squeeze()
        total += flip_labels.size(0)
        correct += (predicted == flip_labels).sum().item()

accuracy = 100 * correct / total
print(f'Accuracy on the test images: {accuracy:.2f}%')
```

```
Accuracy on the test images: 86.40%
```

Accuracy: The overall accuracy of 0.86 suggests that the model correctly predicts the flipped status of images 86% of the time across the test dataset of 10,000 images.

Macro Avg: The macro average for precision, recall, and F1-score (all 0.86) indicates average metric values across classes without considering the imbalance in class distribution.

Weighted Avg: The weighted average also shows 0.86 for precision, recall, and F1-score for class imbalance, showing near balance in class distribution.

	precision	recall	f1-score	support
Not Flipped	0.85	0.88	0.87	5005
Flipped	0.87	0.85	0.86	4995
accuracy			0.86	10000
macro avg	0.86	0.86	0.86	10000
weighted avg	0.86	0.86	0.86	10000

Precision (for each class): The ratio of correctly predicted positive observations to the total predicted positives. The model has a precision of 0.85 for class `0` (not flipped) and 0.87 for class `1` (flipped), meaning it's slightly better at correctly identifying flipped images as flipped than it is at identifying not flipped images as not flipped.

Recall (for each class): The model has a recall of 0.88 for class `0` and 0.85 for class `1`, showing it's slightly better at identifying all not flipped images correctly compared to flipped images.

F1-Score (for each class): The F1-score is the weighted average of Precision and Recall. With F1-scores of 0.87 for class `0` and 0.86 for class `1`, meaning the model has a balanced performance between precision and recall for both classes.

Support: The support is the number of actual occurrences of the class in the specified dataset. There are 5005 not flipped images and 4995 flipped images in the test set.

4.1.2 VGG16

According to the result, we could say that the model performs relatively well and balanced on the binary classification task of predicting whether images from the CIFAR-10 dataset are flipped or not flipped.

```
Epoch 1, Loss: 0.6305108139353335
Epoch 2, Loss: 0.5584006950907086
Epoch 3, Loss: 0.5203515417740473
Epoch 4, Loss: 0.4963480742157573
Epoch 5, Loss: 0.4774317616391975
Epoch 6, Loss: 0.462666368278701
Epoch 7, Loss: 0.447007948404078
Epoch 8, Loss: 0.43210782991041
Epoch 9, Loss: 0.4219830910987256
Epoch 10, Loss: 0.4080733441559555
Finished Training
```

```
model_vgg16.load_state_dict(torch.load(PATH_VGG16))
model_vgg16.eval()

correct = 0
total = 0
test_loader = DataLoader(balanced_test_dataset, batch_size=64, shuffle=False)
with torch.no_grad():
    for images, flip_labels, original_labels in test_loader:
        outputs = model_vgg16(images)
        predicted = torch.round(torch.sigmoid(outputs)).squeeze()
        total += flip_labels.size(0)
        correct += (predicted == flip_labels).sum().item()

accuracy = 100 * correct / total
print(f'Accuracy on the test images: {accuracy:.2f}%')
```

```
Accuracy on the test images: 76.35%
```

Accuracy: The overall accuracy of 0.7635 suggests that the model correctly predicts the flipped status of images 76.35% of the time across the test dataset of 10,000 images.

Macro Avg: The macro average for precision, recall, and F1-score (all 0.76) indicates consistent performance across both classes.

Weighted Avg: The weighted average also shows 0.76 for precision, recall, and F1-score; it takes the support of each class into account, indicating that the performance metrics are nearly balanced across the dataset's distribution.

	precision	recall	f1-score	support
Not Flipped	0.78	0.73	0.75	4951
Flipped	0.75	0.79	0.77	5049
accuracy			0.76	10000
macro avg	0.76	0.76	0.76	10000
weighted avg	0.76	0.76	0.76	10000

Precision (for each class): Out of all the instances predicted as "Not Flipped," 78% were actually "Not Flipped." This means the model is relatively precise in identifying the "Not Flipped" class. The model is slightly less precise for this class “Flipped” with 75%.

Recall (for each class): Of all the actual "Flipped" instances, the model correctly identified 79% of them. The model is better at capturing "Flipped" instances than "Not Flipped" instances (73%).

F1-Score (for each class): The F1-score for "Flipped" is 0.77, slightly higher than for "Not Flipped," indicating a slightly better balance between precision and recall for the "Flipped" class.

Support: The support is the number of actual occurrences of the class in the specified dataset. There are 4951 not flipped images and 5049 flipped images in the test set, only somewhat near balance, compared to the other model.

4.1.3 Concluding Remarks

In conclusion, both models demonstrated commendable performance on the CIFAR-10 dataset, with results that were nearly balanced across most evaluation metrics. However, ResNet18 notably outperformed VGG16, achieving faster training times by an approximate margin of 11%. Several factors may account for this outcome:

- VGG16, being a deeper model with a larger number of parameters than ResNet18, requires more computational resources for training. While this complexity might suggest a **potential for superior performance**, it proved to be **less effective** for this specific task.
- On the bright side, I figured and truly understand how ResNet18 uses residual connections to help mitigate the vanishing gradient problem, allowing for more effective training of deeper networks. These connections make it easier for the network to learn identity mappings.

For VGG16, achieving an overall accuracy of 76% is respectable. Yet, there is more space for further improvement, especially concerning precision and recall for the classified classes.

4.2 Hyper-parameter Tuning

Based on the results, given both models optimized using SGD, ResNet18 outperforms VGG16 in terms of accuracy with significantly faster training time. Given **prolonged training times** and **significant resource consumption** of training VGG16 on Google Colab, I'll focus on hyper-parameter tuning for ResNet18 to see if I can achieve higher result, specifically adjusting its:

- Learning rate controls how quickly or slowly a neural network model learns a problem. Generally, a large learning rate allows the model to learn faster, at the cost of arriving on a sub-optimal final set of weights. [15]
- Training a neural network can be made easier with the addition of history to the weight update. Therefore I also tweaked momentum: *the method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients*. [16]

Adam Optimizer
Instead of tweaking the parameter every time I start training, requiring even more time and resources. Given my limited resource on training data with Google Colab, I decided to try to optimize the model using Adam Optimizer, for the following characteristics which align with my approach above:

- **Adaptive Learning Rates:** Unlike traditional methods that utilize a fixed learning rate or require manual scheduling, Adam adjusts the learning rate on a per-parameter basis [17]. This adaptability allows for more precise and effective model training, potentially leading to improved performance with reduced training times.
- **Momentum Integration:** The Adam optimizer incorporates momentum through its first moment estimate, essentially adding a velocity component to the parameter updates [18]. This not only accelerates training in the right direction but also helps smooth out the updates, making the process more stable and efficient. Also, See PyTorch documentation here.

The training time was significantly reduced from **hours to minutes**. However, the final result is not as good as I expected. The parameter-tuning includes:

1st attempt	2nd attempt	3rd attempt	4th attempt
`lr=0.0005, betas=(0.9, 0.9) weight_decay=1e-8`	`lr=0.0005, betas=(0.85, 0.995) weight_decay=1e-5`	`lr=0.0005, betas=(0.9, 0.99) weight_decay=1e-5`	`lr=0.0002, betas=(0.8, 0.85) weight_decay=1e-5`
61.17% Accuracy	61.84% Accuracy	62.19% Accuracy	62.39% Accuracy

The increase of learning rate, betas, and weight decay have resulted in slight improvements in accuracy across each attempt. However, the overall performance still falls short when compared to expectations and the outcomes achieved using SGD.

In conclusion, for the given dataset and model configuration, SGD appears to be the preferable optimizer compared to Adam.

4.3 Analysis of Misclassifications

Out of the original 10 classes, after the training and testing process, I tried to figure out which 3 classes have the lowest accuracy (less than 86.4 in this case).
To do this, I first implemented a method that:

1. Set up dictionaries to track correct and total predictions for each class:

```
correct_predictions = {label: 0 for label in range(len(class_names))}
total_predictions = {label: 0 for label in range(len(class_names))}
```

2. Loop over test_loader to process each batch of images, flip_labels, and original_labels.

```
   for images, flip_labels, original_labels in test_loader:
       outputs = model_resnet18(images)
       predicted = torch.round(torch.sigmoid(outputs)).squeeze()
```

3. Use the model to predict flip status, compare with true flip_labels, and update accuracy tracking counters.

```
total_predictions[original_labels[i].item()] += 1
           if predicted[i] == flip_labels[i]:
               correct_predictions[original_labels[i].item()] += 1
           else:
               misclassified_info.append((images[i], flip_labels[i],
predicted[i], original_labels[i]))
```

4. Determine class-specific accuracy by dividing the number of correct predictions by the total number of instances for each class.

```
class_accuracy = {class_name: correct_predictions[i] /
total_predictions[i] for i, class_name in enumerate(class_names)}
```

5. Identify Lowest Accuracies: Sort classes by accuracy and select the three with the lowest accuracy percentages.

```
lowest_accuracy_classes = sorted(class_accuracy, key=class_accuracy.get)[:3]
```

The result derived from ResNet18 model

```
Classes with the lowest accuracy in detecting flipped/not flipped:
cat: 73.10%
dog: 80.30%
bird: 80.80%
```

The result derived from VGG16 model

```
Classes with the lowest accuracy in detecting flipped/not flipped:
cat: 63.10%
frog: 63.40%
deer: 69.10%
```

To present 10 images from the wrongly predicted images from the 3 lowest classes, the approach is to:

1. From the misclassified_info list, filter out images from the three classes identified previously.
2. For each of the three classes, randomly select up to 10 misclassified images for display.

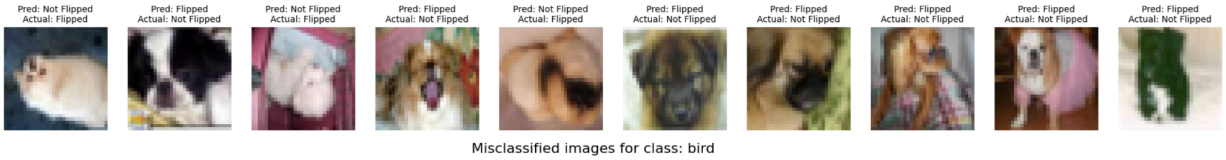
```
lowest_accuracy_class_indices = [class_names.index(class_name) for class_name in
lowest_accuracy_classes]
filtered_misclassified_info = [info for info in misclassified_info if info[3].item()
in lowest_accuracy_class_indices]
```

3. Group misclassified images by their original class label to ensure diversity in the output.

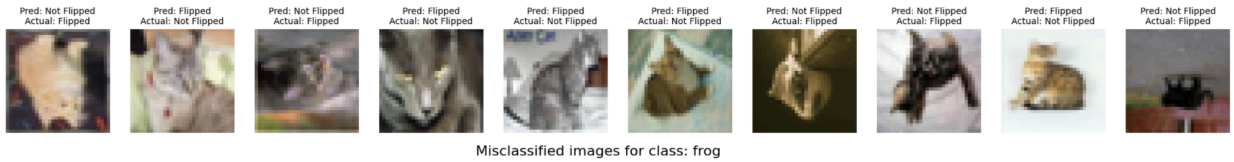
```
misclassified_by_class = defaultdict(list)
for image, flip_status, pred_label, true_label in filtered_misclassified_info:
   misclassified_by_class[true_label.item()].append((image, flip_status, pred_label))
```

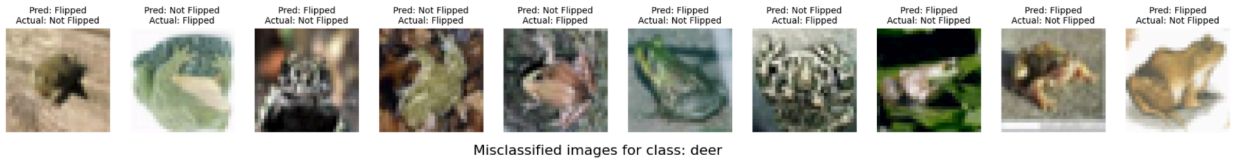
4. For each image, show it along with annotations for actual class, predicted flip status, and whether it was correctly identified as flipped or not flipped.

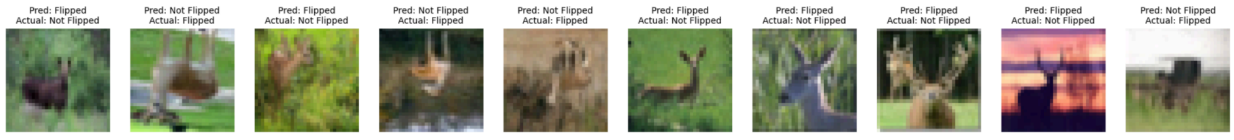
The result derived from ResNet18 model



The result derived from VGG16 model







4.3.1 Error Patterns

4.3.1.1 Common Patterns

The model incorrectly predicted the flipped status for ResNet ["cat," "bird," "dog"] images and VGG16 ["cat," "frog," "deer"], a few common points of these misclassified images may be:

1. Both models struggle the most with accurately classifying "cat" images, with ResNet18 achieving 73.10% and VGG16 significantly lower at 63.10%. This suggests that "cat" images pose a particular challenge, likely because
    a. Cats can significantly vary in color, size, pose, and even fur patterns, more so than many other animals or objects.
    b. Cats may share features with "dog" or even "deer" (e.g., four legs, similar sizes in certain perspectives), causing confusion.
2. These subjects are all animals, so it may not present clear indicators of orientation to the model. Animals can sometimes be similar whether flipped or not.
3. These animals might be captured in poses or orientations that are not typical—like an upside-down bird or a curled-up cat.
4. Factors such as low resolution and blurriness could also make it more difficult to determine whether an image is flipped.

4.3.1.2 Specific challenges of each model

1. ResNet18: The "dog" and "bird" classes are relatively higher in accuracy compared to "cat."
    - "Bird" classification might be affected by the small size of birds in images, varied backgrounds (sky, trees), and their diverse coloration.
    - "Dog" shares some of the variability challenges of "cat," with an extensive range of breeds and appearances.
2. VGG16: "Frog" and "deer" showing low accuracy suggests difficulty in capturing features distinct to these animals, possibly due to:
    - "Frog": Smaller size in images, blended in green backgrounds, and less textured body surfaces.
    - "Deer": Variability in posture, antlers (in males), often blending with natural backgrounds which could be challenging to distinguish from other animals.

4.3.2 Possible Reasons for Errors

Some images were more challenging involves considering several reasons:

1. The model itself might still have **limitations in extracting and interpreting features** that are critical for distinguishing flipped images.

2. The model may **not extract enough relevant features to make accurate predictions** about the flipped status. This could be due to the inherent complexity of the images, such as intricate textures or patterns that do not change significantly when flipped.

3. VGG16, despite its deeper architecture, may not generalize as well across different features or requires more data to overcome its bias towards features learned from ImageNet.

4.3.3 Mitigation Strategies

One of the approaches I could think of is to experiment with **more complex models** or **modifications to the existing architecture** to improve feature extraction capabilities, particularly for textures and orientations.

Secondly, I believe **focused training** would be of great help as it incorporates a targeted subset of training data that specifically addresses the identified weaknesses, such as images with complex backgrounds or unusual subject poses.

Finally, **customizing the model** or training process **for challenging classes**, such as adjusting class weights or employing class-specific networks or layers, might produce better results.

Conclusion

In this project, my exploration spanned several neural network architectures, including ResNet18 and VGG16. I discovered that ResNet18, with its efficient use of residual connections, outperformed the deeper VGG16 model in terms of accuracy and training efficiency. This outcome underscores the significance of **architectural efficiency over depth** in certain image classification tasks.

For hyperparameter tuning, particularly with the **Adam optimizer**, showed that while Adam's adaptive learning rate mechanism offered promise, adjustments to its parameters did not universally translate to performance gains, meaning that the optimizer's advantages might be **more model or task-specific** than previously assumed.

Applying traditional optimizers like SGD, complemented by momentum and weight decay, re-affirmed the value of simplicity and interpretability in achieving high model performance.

Source code

Source codes of this project have been attached to .zip file to Moodle

For a better view option, I have also separated the project into 2 files.

The source code for each model could be found here:

- ResNet18 (applied Adam Optimizer at the end): See here
- VGG16: See here

References

[1] Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.

[2] Simonyan, K., & Zisserman, A. (2015, Apr 10). Very deep convolutional networks for large-scale image recognition. arXiv. https://doi.org/10.48550/arXiv.1409.1556v6

[3] Adam Coates, Honglak Lee, Andrew Y. Ng An Analysis of Single Layer Networks in Unsupervised Feature Learning AISTATS, 2011.

[4] Chng, Z. M. (2022, June 20). *Using normalization layers to improve deep learning models*. Machine Learning Mastery. https://machinelearningmastery.com/using-normalization-layers-to-improve-deep-learning-models/

[5] Karakus, B. (2020, December 30). I am trying to understand how the process of calculating the mean for every channel of RGB image. Stack Overflow. https://stackoverflow.com/questions/65501984/i-am-trying-to-understand-how-the-process-of-calculating-the-mean-for-every-chan

[6] Sundararajan, B. (2024, January 21). *Decoding ResNet: Revolutionizing deep learning architectures*. Medium. https://medium.com/@bragadeeshs/decoding-resnet-revolutionizing-deep-learning-architectures-19faf9658fa4

[7] Purakkatt, A. (2020, October 21). ResNet. *Analytics Vidhya*. https://medium.com/analytics-vidhya/resnet-10f4ef1b9d4c

[8] Thakur, R. (2024, March 12). VGG16. Built In. Updated by Brennan Whitfield. https://builtin.com/machine-learning/vgg16

[9] Vaj, T. (2024, January 24). Why we freeze some layers for transfer learning. Medium. https://vtiya.medium.com/why-we-freeze-some-layers-for-transfer-learning-f35d9f67f99c

[10] Alberto, C. (2018, May 4). VGG16 transfer learning PyTorch. Kaggle. https://www.kaggle.com/code/carloalbertobarbano/vgg16-transfer-learning-pytorch

[11] Carter, S. (2023, December 27). Why nn.BCEWithLogitsLoss is numerically stable. Medium. https://medium.com/@sahilcarterr/why-nn-bcewithlogitsloss-numerically-stable-6a04f3052967

[12] ML Cheatsheet. (n.d.). Loss functions. Retrieved on March 20, from https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html

[13] Topper, N. (2023, July 10). Sigmoid activation function. Built In. https://builtin.com/machine-learning/sigmoid-activation-function

[14] Chaudhary, K. (2023, November 15). Understanding optimizers for training deep learning models. Medium. https://medium.com/game-of-bits/understanding-optimizers-for-training-deep-learning-models-694c071b5b70

[15] Brownlee, J. (2019, August 6). Learning rate for deep learning neural networks. Machine Learning Mastery. https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/

[16] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press. (p. 296).

[17] Devapatla, K. K. (2023, March 30). Getting to know Adam optimization: A comprehensive guide. LinkedIn. https://www.linkedin.com/pulse/getting-know-adam-optimization-comprehensive-guide-kiran-kumar

[18] Ruder, S. (2017, June 15). An overview of gradient descent optimization algorithms. arXiv. https://doi.org/10.48550/arXiv.1609.04747v2