Project 2 - Anomaly Detection in caltech101

Group 5: Yin Yin Thu, Wiwat Pholsomboon, Wassim Mecheri

Introduction

- Detect non-airplane (anomaly) images in Caltech101 dataset
- Vanilla Autoencoder will be used to detect anomalies (non-airplane images) from normal class (airplane)
- VAE and -VAE will be test as alternative model to improve the performance of the detection.

Data Preprocessing and Exploration

Import libraries

[1]:
```
# Import libraries
import tensorflow_datasets as tfds
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from tensorflow import keras
from tensorflow.keras import layers
from tabulate import tabulate
from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_score,␣
 ↪accuracy_score, roc_curve, auc

BATCH_SIZE = 64
RANDOM_SEED = 555
IMAGE_SIZE = 96
LATENT_DIM = 512
THRESHOLD = 0.03
VERBOSE = 0

tf.random.set_seed(RANDOM_SEED)
```

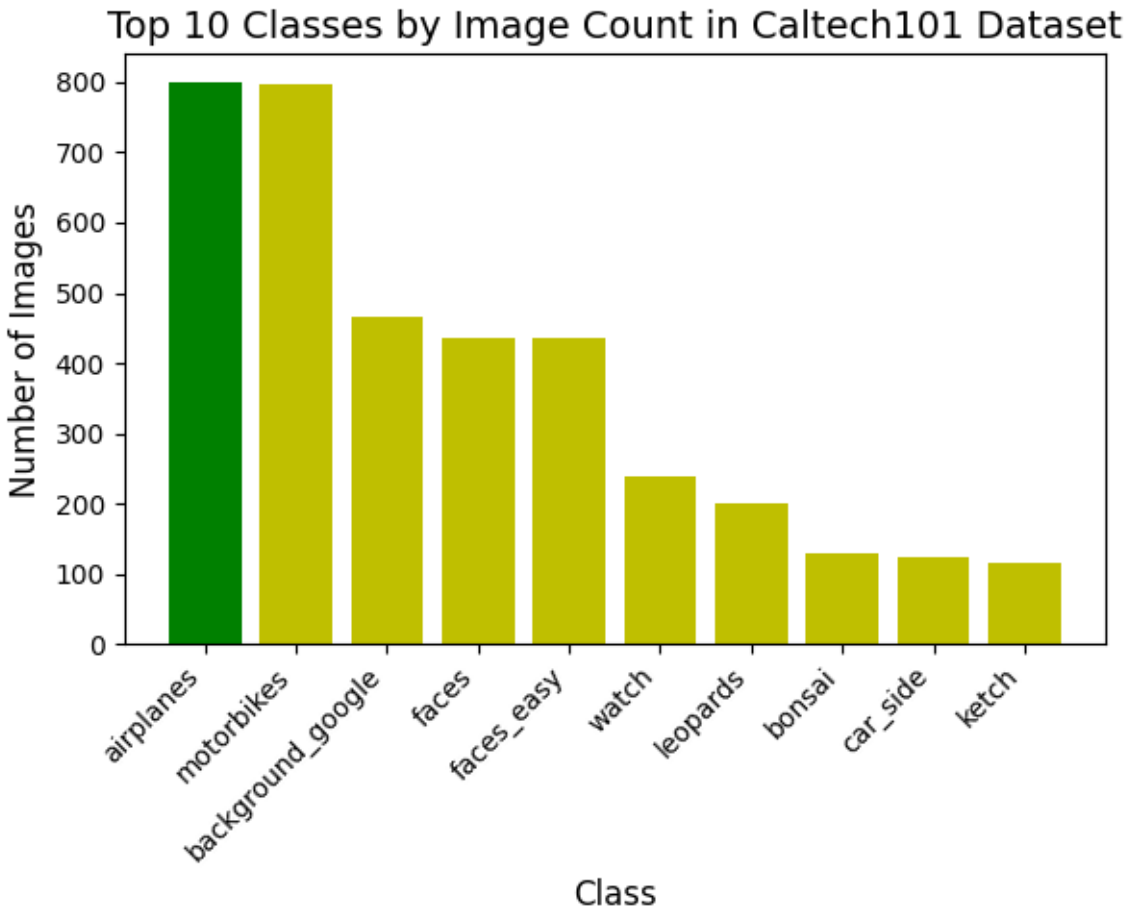
Load the Caltech101 dataset

- Images dataset which contain 102 classes.
- Visualize the distribution of classes in the dataset

[2]:
```
# Load and combine datasets
ds = tfds.load('caltech101', split='train+test', as_supervised=True)

# Count images per class
class_counts = {}
for _, label in ds:
    label_id = label.numpy()
    class_counts[label_id] = class_counts.get(label_id, 0) + 1

# Get class names
class_names = tfds.builder('caltech101').info.features['label'].names

# Create and sort DataFrame
```

```
df = pd.DataFrame({
    'class_name': [class_names[i] for i in class_counts.keys()],
    'count': list(class_counts.values())
})
top10_df = df.nlargest(10, 'count')

# Plot
plt.figure(figsize=(6, 5))
plt.bar(top10_df['class_name'], top10_df['count'], color=['g', 'y', 'y', 'y', 'y',␣
 ↪'y', 'y', 'y', 'y', 'y'])
plt.xlabel('Class', fontsize=12)
plt.ylabel('Number of Images', fontsize=12)
plt.title('Top 10 Classes by Image Count in Caltech101 Dataset', fontsize=14)
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()

# Print summary
print(f"Total classes in dataset: {len(class_names)}")
print(f"Total images in dataset: {sum(class_counts.values())}")
```

```
2025-03-28 14:30:05.707306: I metal_plugin/src/device/metal_device.cc:1154] Metal device
set to: Apple M3 Max
2025-03-28 14:30:05.707333: I metal_plugin/src/device/metal_device.cc:296] systemMemory:
36.00 GB
2025-03-28 14:30:05.707337: I metal_plugin/src/device/metal_device.cc:313] maxCacheSize:
13.50 GB
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1743186605.707351 9344565 pluggable_device_factory.cc:305] Could not identify
NUMA node of platform GPU ID 0, defaulting to 0. Your kernel may not have been built with
NUMA support.
I0000 00:00:1743186605.707366 9344565 pluggable_device_factory.cc:271] Created TensorFlow
device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical
PluggableDevice (device: 0, name: METAL, pci bus id: <undefined>)
2025-03-28 14:30:05.767368: I tensorflow/core/kernels/data/tf_record_dataset_op.cc:376]
The default buffer size is 262144, which is overridden by the user specified
`buffer_size` of 8388608
2025-03-28 14:30:06.360263: I tensorflow/core/framework/local_rendezvous.cc:405] Local
rendezvous is aborting with status: OUT_OF_RANGE: End of sequence
```




```
Total classes in dataset: 102
Total images in dataset: 9144
```

Preprocessing

- Resize the image to 96x96 and preserve the aspect ratio
- Center crop the image to 96x96
- Normalize the image pixel values to 0-1
- Split the dataset into normal, anomaly images and evaluation dataset

[3]:
```python
### Split data into normal, anomaly sand evauiondataset
def scale_resize_image(image, label):
    image = tf.cast(image, tf.float32) / 255.0
    # Get center crop with aspect ratio preserved
    min_dim = tf.minimum(tf.shape(image)[0], tf.shape(image)[1])
    image = tf.image.crop_to_bounding_box(
        image,
        (tf.shape(image)[0] - min_dim) // 2,
        (tf.shape(image)[1] - min_dim) // 2,
        min_dim,
        min_dim
    )
```

```
    image = tf.image.resize(image, [IMAGE_SIZE, IMAGE_SIZE], ) # Resizing the image to␣
 ↪224x224 dimention
    return (image, label)

# Resize and center crop the image to 96x96 and convert to numpy array for␣
 ↪visualization
combined_ds = (ds.map(scale_resize_image)).cache()
combined_images = [(image, label == 1) for image, label in combined_ds]

# Split 3 dataset
# For training and visualization
normal_images = np.array([image for image, airplane in combined_images if airplane ==␣
 ↪True ])
anomaly_images = np.array([image for image, airplane in combined_images if airplane ==␣
 ↪False ])

# For evaluation
test_X = np.array([image for (image, _) in combined_images])
test_y = np.array([0 if airplane == True else 1 for (_, airplane) in combined_images])
```

```
2025-03-28 14:30:07.187490: I tensorflow/core/framework/local_rendezvous.cc:405] Local
rendezvous is aborting with status: OUT_OF_RANGE: End of sequence
```

Exploration

- Show sample images from normal (Airplane) and anomaly (Non-Airplane) classes

[4]:
```
# Display sample images from specific classes
def show_class_samples(dataset, class_name, num_samples=8):

    # Take the specified number of samples
    samples = dataset[:num_samples]

    # Create a figure to display the images
    fig, axes = plt.subplots(1, num_samples, figsize=(10, 2))
    fig.suptitle(f'Sample images from class: {class_name}', fontsize=14)

    # Display each sample
    for i, image in enumerate(samples):
        if i < num_samples:
            axes[i].imshow(image)
            axes[i].axis('off')

    plt.tight_layout()
    plt.show()

# Show samples from class 1 (airplane)
show_class_samples(normal_images, 'Normal Images')
show_class_samples(anomaly_images, 'Anomaly Images')
```

Sample images from class: Normal Images

Sample images from class: Anomaly Images

Unsupervised Learning Model Development

Vanilla Autoencoder

- Encoder contains 3 convolutional layers and 1 fully connected layer convert to latent vector
- Decoder contains 3 convolutional transpose layers and 1 convolutional layer to reconstruct the image

[5]:
```
# Build the Autoencoder model
class Autoencoder(keras.Model):
    def __init__(self, img_shape, latent_dim, num_hidden_layer, num_filter, **kwargs):
        super(Autoencoder, self).__init__(**kwargs)

        encoder = tf.keras.Sequential()
        for _ in range(num_hidden_layer):
            encoder.add(layers.Conv2D(num_filter, 3, activation="relu", strides=2,␣
↪padding="same", kernel_initializer=tf.keras.initializers.
↪GlorotUniform(seed=RANDOM_SEED)))
            num_filter *= 2
        encoder.add(layers.Flatten())
        encoder.add(layers.Dense(num_filter * (2 ** num_hidden_layer),␣
↪activation="relu", kernel_initializer=tf.keras.initializers.
↪GlorotUniform(seed=RANDOM_SEED)))
        encoder.add(layers.Dense(latent_dim, name="latent_vector",␣
↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)))

        h, w, c = img_shape
        decoder_starting_dims = (h // (2 ** num_hidden_layer), w // (2 **␣
↪num_hidden_layer), 128)

        decoder = tf.keras.Sequential()
        decoder.add(layers.Dense(np.prod(decoder_starting_dims), activation="relu",␣
↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)))
        decoder.add(layers.Reshape(decoder_starting_dims))
```

```
        for _ in range(num_hidden_layer):
            decoder.add(layers.Conv2DTranspose(num_filter, 3, activation="relu",␣
 ↪strides=2, padding="same", kernel_initializer=tf.keras.initializers.
 ↪GlorotUniform(seed=RANDOM_SEED)))
            num_filter //= 2
        decoder.add(layers.Conv2D(c, 3, activation="sigmoid", padding="same",␣
 ↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)))

        self.encoder = encoder
        self.decoder = decoder

    def call(self, inputs):
        latent = self.encoder(inputs)
        return self.decoder(latent)

# Main training function
def train_autoencoder(dataset, img_shape=(224, 224, 3), latent_dim=64, epochs=20,␣
 ↪learning_rate=0.001, num_hidden_layer=3, num_filter=32):
    # Create Autoencoder
    autoencoder = Autoencoder(img_shape, latent_dim, num_hidden_layer, num_filter)
    autoencoder.compile(optimizer=keras.optimizers.Adam(learning_rate=learning_rate),␣
 ↪loss="mse", metrics=["mse"])
    # Train the model
    history = autoencoder.fit(dataset, dataset, epochs=epochs, batch_size=BATCH_SIZE,␣
 ↪verbose=VERBOSE)

    return autoencoder, history

# Generate reconstructions from the autoencoder
def generate_reconstructions(autoencoder, test_images, n=8):
    # Get test images
    test_sample = test_images[:n]
    # Get reconstructions
    reconstructed = autoencoder.predict(test_sample, verbose=VERBOSE)

    return test_sample, reconstructed

# Visualize original vs reconstructed images
def visualize_reconstructions(originals, reconstructions, n=8):
    import matplotlib.pyplot as plt

    plt.figure(figsize=(14, 4))
    for i in range(n):
        # Original
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(originals[i])
        plt.title("Original")
        plt.axis("off")

        # Reconstruction
        ax = plt.subplot(2, n, i + 1 + n)
```

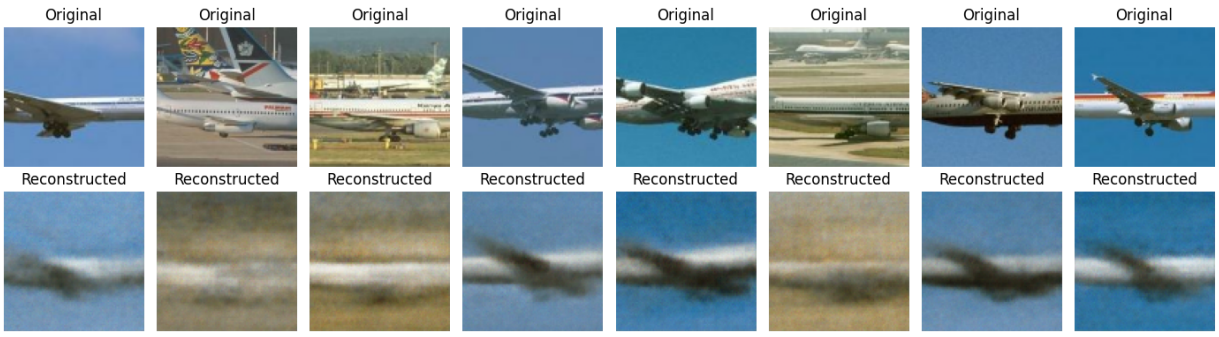
```
        plt.imshow(reconstructions[i])
        plt.title("Reconstructed")
        plt.axis("off")

    plt.tight_layout()
    plt.show()

# Calculate ROC AUC score for both models
def calculate_roc_auc(y_true, scores):
    # Normalize scores to 0-1 range
    scores_normalized = (scores - np.min(scores)) / (np.max(scores) - np.min(scores))
    fpr, tpr, _ = roc_curve(y_true, scores_normalized)
    return fpr, tpr, auc(fpr, tpr)
```

Train the Autoencoder and generate reconstructions

- Train the autoencoder with normal images (Airplane)
- Generate reconstructions from the autoencoder
- Loss function is Mean Squared Error (MSE)
- Optimizer is Adam with learning rate 0.001

[6]:
```
# Train the autoencoder
autoencoder, history = train_autoencoder(normal_images, img_shape=(IMAGE_SIZE,␣
 ↪IMAGE_SIZE, 3), latent_dim=LATENT_DIM, epochs=20)

# Generate reconstructions
original, reconstructed = generate_reconstructions(autoencoder, normal_images, n=8)
visualize_reconstructions(original, reconstructed, n=8)
```

```
2025-03-28 14:30:08.766459: I
tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:117] Plugin
optimizer for device_type GPU is enabled.
```

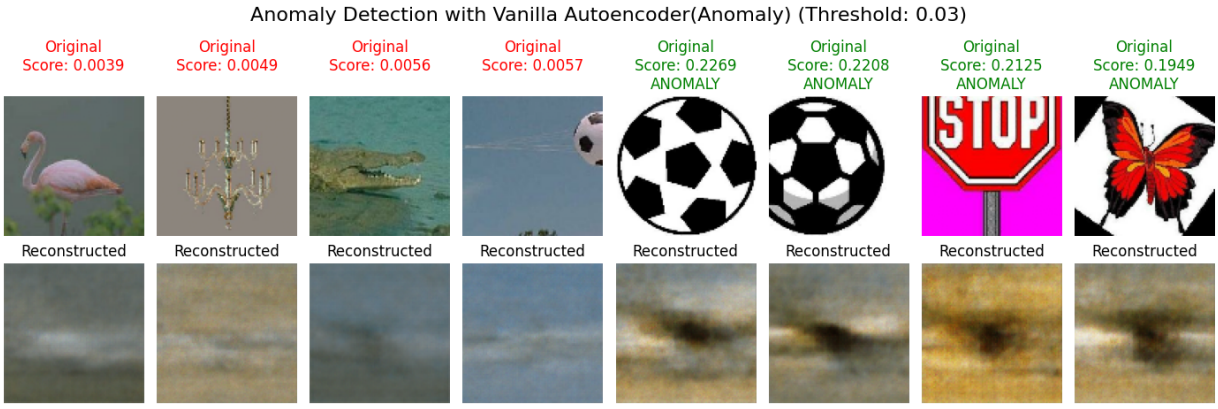


Anomaly Detection

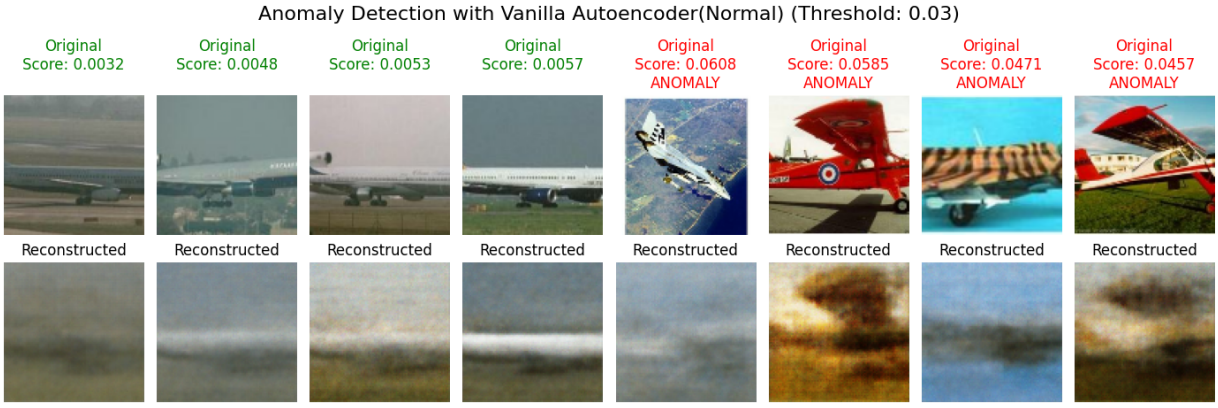
- Detect anomalies (image from other classes) with trainied autoencoder.
- Visualize the best and the worst images from anomaly classes and normal classs.
- Threshold is 0.03, come from hyperparameter tuning result.

[7]:
```
# Function to detect anomalies using autoencoder
def detect_anomalies_autoencoder(autoencoder, images, threshold):
    reconstructions = autoencoder.predict(images, verbose=VERBOSE)
    # Calculate MSE for each image
    mse = np.mean(np.square(images - reconstructions), axis=(1, 2, 3))
    # Determine if each image is an anomaly based on threshold
    anomalies = mse > threshold
    return mse, anomalies, reconstructions

# Function to visualize anomalies
def visualize_anomalies(original_images, reconstructions, scores, anomalies,␣
 ↪model_name, n=8, is_normal_class=False):
    plt.figure(figsize=(14, 5))
    # Get indices with lowest and highest scores
    display_indices = np.concatenate([np.argsort(scores)[:n//2], np.argsort(scores)[::
 ↪-1][:n//2]])

    # Set colors based on class type
    anomaly_color = 'red' if is_normal_class else 'green'
    normal_color = 'green' if is_normal_class else 'red'

    for i in range(min(n, len(original_images))):
        idx = display_indices[i]
        # Original image
        plt.subplot(2, n, i + 1)
        plt.imshow(original_images[idx])
        title = f"Original\nScore: {scores[idx]:.4f}"
        title += "\nANOMALY" if anomalies[idx] else "\n"
        plt.title(title, color=anomaly_color if anomalies[idx] else normal_color)
        plt.axis("off")

        # Reconstruction
        plt.subplot(2, n, i + n + 1)
        plt.imshow(reconstructions[idx])
        plt.title("Reconstructed")
        plt.axis("off")

    plt.suptitle(f"Anomaly Detection with {model_name} (Threshold: {THRESHOLD})",␣
 ↪fontsize=16)
    plt.tight_layout()
    plt.show()

# Detect anomalies with regular autoencoder
ae_scores, ae_anomalies, ae_reconstructions =␣
 ↪detect_anomalies_autoencoder(autoencoder, anomaly_images, threshold=THRESHOLD)
visualize_anomalies(anomaly_images, ae_reconstructions, ae_scores, ae_anomalies,␣
 ↪"Vanilla Autoencoder(Anomaly)", is_normal_class=False)

print(f"Vanilla Autoencoder detected {np.sum(ae_anomalies)} anomalies from␣
 ↪{len(anomaly_images)} images")
```

```
# Detect anomalies with regular autoencoder
normal_scores, normal_anomalies, normal_reconstructions =␣
 ↪detect_anomalies_autoencoder(autoencoder, normal_images, threshold=THRESHOLD)
visualize_anomalies(normal_images, normal_reconstructions, normal_scores,␣
 ↪normal_anomalies, "Vanilla Autoencoder(Normal)", is_normal_class=True)

print(f"Vanilla Autoencoder disclassified {np.sum(normal_anomalies)} normal images␣
 ↪from {len(normal_images)} images")
```




```
Vanilla Autoencoder detected 6632 anomalies from 8344 images
```




```
Vanilla Autoencoder disclassified 43 normal images from 800 images
```

Actionable Recommendations

Feature Analysis

- Feature analysis below show the portion of the image that contribute to the anomaly detection.

1. Anomaly Item detected as Normal(Airplane)

- Has background color as grey or blue.
- Has object on the center of the image.

- Majority of the image is background.

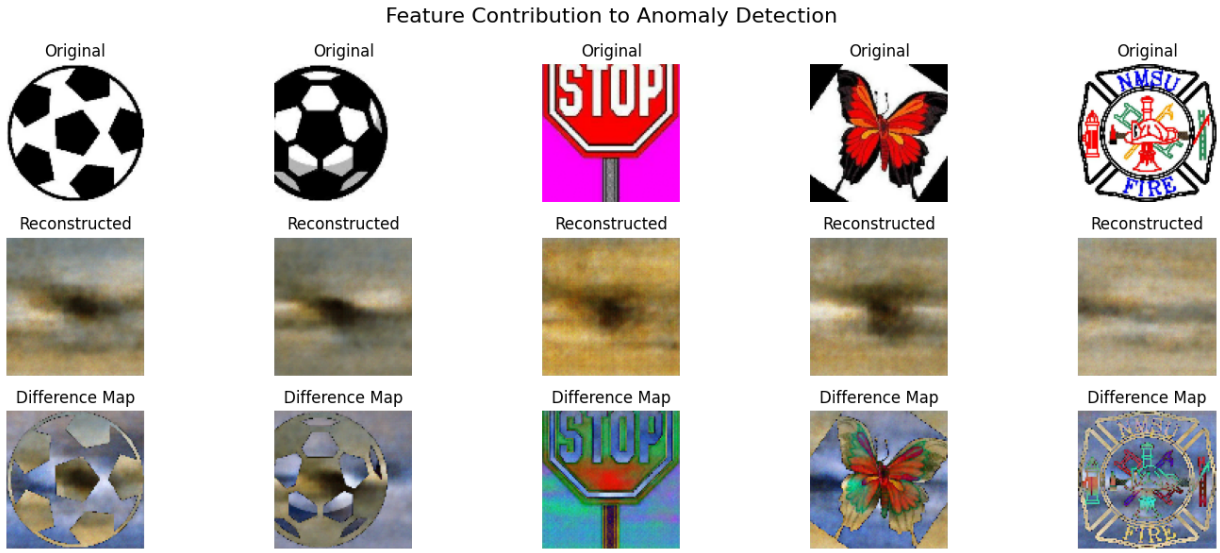
2. **Normal Item detected as Anomaly(Non-Airplane)**

- Plane is in red color and cover more than 50% of the image.
- Plane which station on the ground and the background is not sky.

Actionable Recommendations

1. Increase complexity of the model to detect more patterns.
2. Increase size of dataset use to train the model. Especially on red airplane and airplane station on the ground.
3. Increase latent dimension to capture more features.

[8]:
```python
# Actionable Recommendations (4 marks)

## Feature Importance Analysis
# Function to visualize the difference between original and reconstructed images
def visualize_difference_maps(originals, reconstructions, n=5):
    plt.figure(figsize=(15, 6))
    for i in range(min(n, len(originals))):
        # Original
        plt.subplot(3, n, i + 1)
        plt.imshow(originals[i])
        plt.title("Original")
        plt.axis("off")

        # Reconstruction
        plt.subplot(3, n, i + 1 + n)
        plt.imshow(reconstructions[i])
        plt.title("Reconstructed")
        plt.axis("off")

        # Difference map
        diff = np.abs(originals[i] - reconstructions[i])
        # Normalize for better visualization
        diff = diff / np.max(diff) if np.max(diff) > 0 else diff

        plt.subplot(3, n, i + 1 + 2*n)
        plt.imshow(diff, cmap='hot')
        plt.title("Difference Map")
        plt.axis("off")

    plt.suptitle("Feature Contribution to Anomaly Detection", fontsize=16)
    plt.tight_layout()
    plt.show()

# Select some high-anomaly score examples
anomaly_indices = np.argsort(-ae_scores)[:5]  # Top 5 highest anomaly scores
high_anomaly_imgs = anomaly_images[anomaly_indices]
high_anomaly_recon = ae_reconstructions[anomaly_indices]
visualize_difference_maps(high_anomaly_imgs, high_anomaly_recon)
```




```
Bright color in Difference Map indicated area that is different between original and reconstruct
```

Evaluation and Visualization

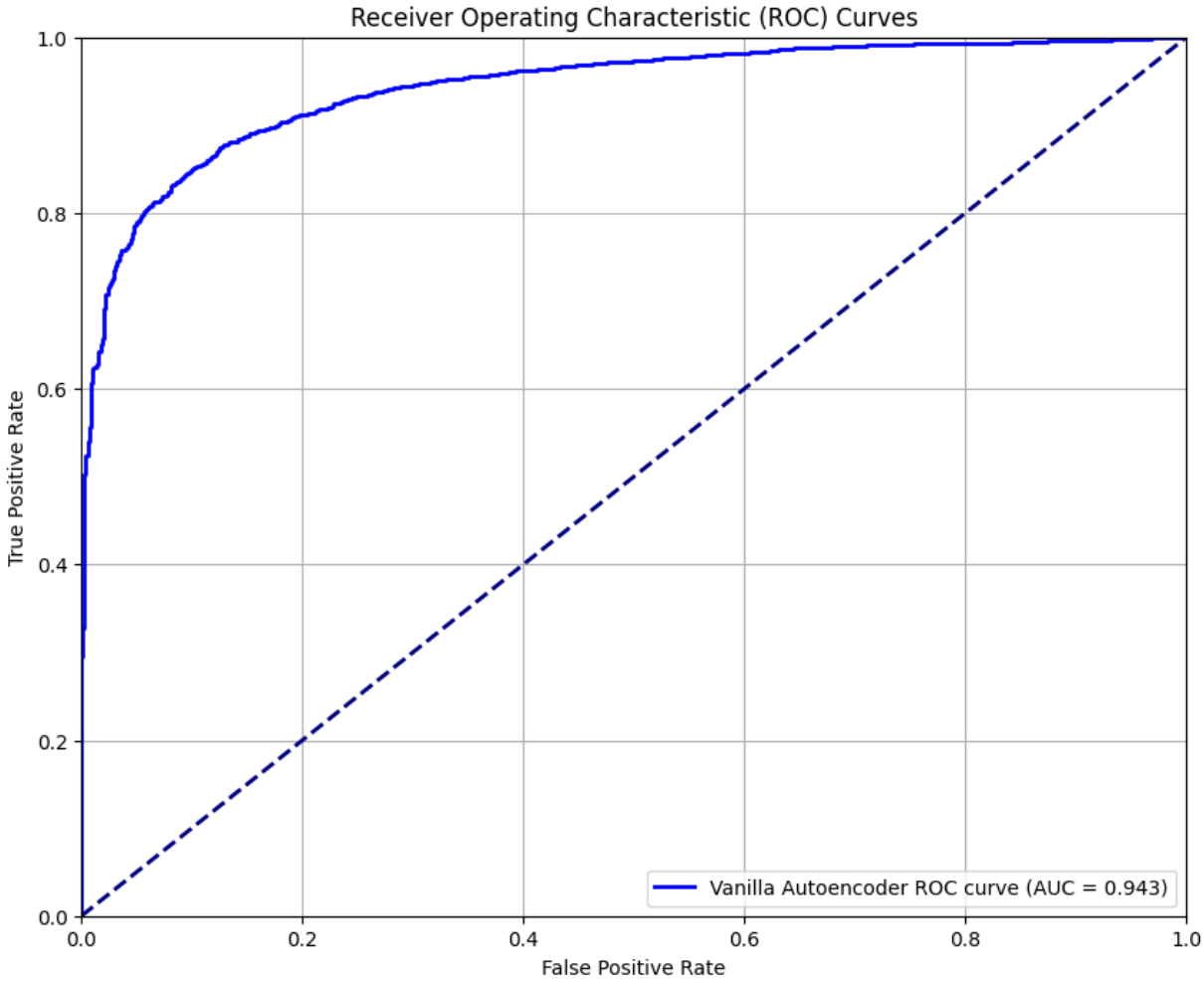
- Calculate evaluation metrics for vanilla autoencoder.
- Plot ROC/AUC curve vanilla autoencoder.

[9]:
```python
def calculate_metrics(y_true, scores, threshold):
    # Convert scores to predictions based on threshold
    y_pred = (scores > threshold).astype(int)

    # Calculate basic metrics
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)

    # Calculate confusion matrix
    tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()

    return [
        ["Metric", "Value"],
        ["Accuracy", f"{accuracy:.4f}"],
        ["Precision", f"{precision:.4f}"],
        ["Recall", f"{recall:.4f}"],
        ["F1 Score", f"{f1:.4f}"],
        ["  True Positives", tp],
        ["  False Positives", fp],
        ["  True Negatives", tn],
        ["  False Negatives", fn]
    ]

# Calculate metrics for both models on the test set
```

```
print("Calculating performance metrics for Vanilla Autoencoder...")
ae_scores, ae_anomalies, ae_reconstructions =␣
 ↪detect_anomalies_autoencoder(autoencoder, test_X, THRESHOLD)
ae_metrics = calculate_metrics(test_y, ae_scores, THRESHOLD)

print("\nAutoencoder Performance Metrics:")
print(tabulate(ae_metrics, headers="firstrow", tablefmt="grid", numalign="right"))

# Calculate ROC AUC score
fpr_ae, tpr_ae, auc_ae = calculate_roc_auc(test_y, ae_scores)

# Plot ROC curves
plt.figure(figsize=(10, 8))
plt.plot(fpr_ae, tpr_ae, color='blue', lw=2, label=f'Vanilla Autoencoder ROC curve␣
 ↪(AUC = {auc_ae:.3f})')

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curves')
plt.legend(loc="lower right")
plt.grid(True)
plt.show()
```

```
Calculating performance metrics for Vanilla Autoencoder…

Autoencoder Performance Metrics:
```

Metric	Value
Accuracy	0.8081
Precision	0.9936
Recall	0.7948
F1 Score	0.8831
True Positives	6632
False Positives	43
True Negatives	757
False Negatives	1712



Model Refinement and Optimization

List of hyperparameter tuning and alternative model finding: 1. Hyperparameter tuning on Vanilla Autoencoder in various threshold. 2. Hyperparameter tuning on Vanilla Autoencoder in various latent dimension. 3. Hyperparameter tuning on Vanilla Autoencoder in various number of hidden layer. 4. Try to use VAE to improve the performance of the detection. 5. Try to use -VAE to improve the performance of the detection.

1. Hyperparameter Tuning on Vanilla Autoencoder in various threshold

- Threshold `0.01` has the highest f1 score and accuracy indicating best performance but it miss a lot on normal class.
- Threshold `0.03` show balanced performance on both normal and anomaly class.
- By the way, in real life scenario, we might need to consider the cost of false alarm and false negative on Airplane detection which might make `0.04` or `0.05` the best threshold if Airplant miss detection is costly.

[10]:
```
thresholds = [0.005, 0.01, 0.02, 0.03, 0.04, 0.05]

# Store metrics for each threshold
all_metrics = {}
```

```python
for threshold in thresholds:
    # Get metrics for current threshold
    metrics = calculate_metrics(test_y, ae_scores, threshold)

    # Extract metric names and values (skip header row)
    metric_data = metrics[1:]

    # Initialize dictionary on first run
    if not all_metrics:
        all_metrics = {row[0]: [] for row in metric_data}

    # Add values for this threshold
    for row in metric_data:
        all_metrics[row[0]].append(row[1])

# Build the table
header_row = ["Metric"] + [f"{t}" for t in thresholds]
combined_table = [header_row]

# Add each metric row
for metric, values in all_metrics.items():
    combined_table.append([metric] + values)

# Display results
print("Vanilla Autoencoder Performance Across Thresholds:")
print(tabulate(combined_table, headers="firstrow", tablefmt="grid", numalign="right"))
```

```
Vanilla Autoencoder Performance Across Thresholds:
+-------------------+---------+--------+--------+--------+--------+--------+
| Metric            |   0.005 |   0.01 |   0.02 |   0.03 |   0.04 |   0.05 |
+===================+=========+========+========+========+========+========+
| Accuracy          |  0.9125 | 0.9228 | 0.9132 | 0.8081 |  0.659 | 0.5052 |
+-------------------+---------+--------+--------+--------+--------+--------+
| Precision         |  0.9127 | 0.9274 |  0.976 | 0.9936 | 0.9977 | 0.9995 |
+-------------------+---------+--------+--------+--------+--------+--------+
| Recall            |  0.9998 | 0.9932 | 0.9276 | 0.7948 | 0.6278 | 0.4581 |
+-------------------+---------+--------+--------+--------+--------+--------+
| F1 Score          |  0.9542 | 0.9591 | 0.9512 | 0.8831 | 0.7706 | 0.6282 |
+-------------------+---------+--------+--------+--------+--------+--------+
|   True Positives  |    8342 |   8287 |   7740 |   6632 |   5238 |   3822 |
+-------------------+---------+--------+--------+--------+--------+--------+
|   False Positives |     798 |    649 |    190 |     43 |     12 |      2 |
+-------------------+---------+--------+--------+--------+--------+--------+
|   True Negatives  |       2 |    151 |    610 |    757 |    788 |    798 |
+-------------------+---------+--------+--------+--------+--------+--------+
|   False Negatives |       2 |     57 |    604 |   1712 |   3106 |   4522 |
+-------------------+---------+--------+--------+--------+--------+--------+
```

2. Hyperparameter tuning on Vanilla Autoencoder in various latent dimension.

- Latent dimention 1280 show the best performance of F1-score balanced between precision and recall.
- Latent dimention 1024 show best performance on normal class make it most suitable for real life

scenario.

[11]:
```
latent_dims = [512, 768, 1024, 1280]

# Store metrics for each threshold
all_metrics = {}

for latent_dim in latent_dims:
    # Get metrics for current threshold
    autoencoder, history = train_autoencoder(normal_images, img_shape=(IMAGE_SIZE,␣
 ↪IMAGE_SIZE, 3), latent_dim=latent_dim, epochs=20)
    ae_scores, ae_anomalies, ae_reconstructions =␣
 ↪detect_anomalies_autoencoder(autoencoder, test_X, THRESHOLD)
    metrics = calculate_metrics(test_y, ae_scores, THRESHOLD)

    # Extract metric names and values (skip header row)
    metric_data = metrics[1:]

    # Initialize dictionary on first run
    if not all_metrics:
        all_metrics = {row[0]: [] for row in metric_data}

    # Add values for this threshold
    for row in metric_data:
        all_metrics[row[0]].append(row[1])

# Build the table
header_row = ["Metric"] + [f"{t}" for t in latent_dims]
combined_table = [header_row]

# Add each metric row
for metric, values in all_metrics.items():
    combined_table.append([metric] + values)

# Display results
print("Vanilla Autoencoder Performance Across Latent Dimension:")
print(tabulate(combined_table, headers="firstrow", tablefmt="grid", numalign="right"))
```

```
Vanilla Autoencoder Performance Across Latent Dimension:
+-------------------+--------+--------+--------+--------+
| Metric            |    512 |    768 |   1024 |   1280 |
+===================+========+========+========+========+
| Accuracy          | 0.8081 | 0.8109 | 0.7818 | 0.8108 |
+-------------------+--------+--------+--------+--------+
| Precision         | 0.9936 | 0.9954 | 0.9967 | 0.9898 |
+-------------------+--------+--------+--------+--------+
| Recall            | 0.7948 | 0.7965 | 0.7634 | 0.8009 |
+-------------------+--------+--------+--------+--------+
| F1 Score          | 0.8831 | 0.8849 | 0.8646 | 0.8854 |
+-------------------+--------+--------+--------+--------+
|   True Positives  |   6632 |   6646 |   6370 |   6683 |
+-------------------+--------+--------+--------+--------+
```

```
|   False Positives |     43 |     31 |     21 |     69 |
+-------------------+--------+--------+--------+--------+
|   True Negatives  |    757 |    769 |    779 |    731 |
+-------------------+--------+--------+--------+--------+
|   False Negatives |   1712 |   1698 |   1974 |   1661 |
+-------------------+--------+--------+--------+--------+
```

3. Hyperparameter tuning on Vanilla Autoencoder in various number of hidden layer.

- Number of hidden layer 5 show the best performance of F1-score balanced between precision and recall.
- Number of hidden layer 3 show best performance on normal class make it most suitable for real life scenario.

[12]:
```
hidden_layers = [3, 4, 5]

# Store metrics for each threshold
all_metrics = {}

for hidden_layer in hidden_layers:
    # Get metrics for current threshold
    autoencoder, history = train_autoencoder(normal_images, img_shape=(IMAGE_SIZE,␣
 ↪IMAGE_SIZE, 3), num_hidden_layer=hidden_layer, epochs=20)
    ae_scores, ae_anomalies, ae_reconstructions =␣
 ↪detect_anomalies_autoencoder(autoencoder, test_X, THRESHOLD)
    metrics = calculate_metrics(test_y, ae_scores, THRESHOLD)

    # Extract metric names and values (skip header row)
    metric_data = metrics[1:]

    # Initialize dictionary on first run
    if not all_metrics:
        all_metrics = {row[0]: [] for row in metric_data}

    # Add values for this threshold
    for row in metric_data:
        all_metrics[row[0]].append(row[1])

# Build the table
header_row = ["Metric"] + [f"{t}" for t in hidden_layers]
combined_table = [header_row]

# Add each metric row
for metric, values in all_metrics.items():
    combined_table.append([metric] + values)

# Display results
print("Vanilla Autoencoder Performance Across Hidden Layer:")
print(tabulate(combined_table, headers="firstrow", tablefmt="grid", numalign="right"))
```

```
Vanilla Autoencoder Performance Across Hidden Layer:
+-------------------+--------+--------+--------+
| Metric            |      3 |      4 |      5 |
+===================+========+========+========+
```

```
| Accuracy           | 0.7772 |  0.807 | 0.8759 |
+--------------------+--------+--------+--------+
| Precision          | 0.9964 | 0.9913 | 0.9818 |
+--------------------+--------+--------+--------+
| Recall             | 0.7586 | 0.7954 | 0.8803 |
+--------------------+--------+--------+--------+
| F1 Score           | 0.8614 | 0.8826 | 0.9283 |
+--------------------+--------+--------+--------+
|   True Positives   |   6330 |   6637 |   7345 |
+--------------------+--------+--------+--------+
|   False Positives  |     23 |     58 |    136 |
+--------------------+--------+--------+--------+
|   True Negatives   |    777 |    742 |    664 |
+--------------------+--------+--------+--------+
|   False Negatives  |   2014 |   1707 |    999 |
+--------------------+--------+--------+--------+
```

4. Variational Autoencoder

- define function to train, visualize, and detect anomalies for VAE

[13]:
```
# Build the VAE model
class VAE(keras.Model):
    def __init__(self, img_shape, latent_dim, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = keras.Sequential(
            [
                layers.Conv2D(32, 3, activation="relu", strides=2, padding="same",␣
 ↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)),
                layers.Conv2D(64, 3, activation="relu", strides=2, padding="same",␣
 ↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)),
                layers.Conv2D(128, 3, activation="relu", strides=2, padding="same",␣
 ↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)),
                layers.Flatten(),
                layers.Dense(256, activation="relu", kernel_initializer=tf.keras.
 ↪initializers.GlorotUniform(seed=RANDOM_SEED)),
                layers.Dense(latent_dim * 2, name="latent_vector",␣
 ↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)),
            ]
        )

        h, w, c = img_shape
        encoder_conv_layers = 3
        decoder_starting_dims = (h // (2 ** encoder_conv_layers), w // (2 **␣
 ↪encoder_conv_layers), 128)

        self.decoder = keras.Sequential(
            [
                layers.Dense(np.prod(decoder_starting_dims), activation="relu",␣
 ↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)),
                layers.Reshape(decoder_starting_dims),
```

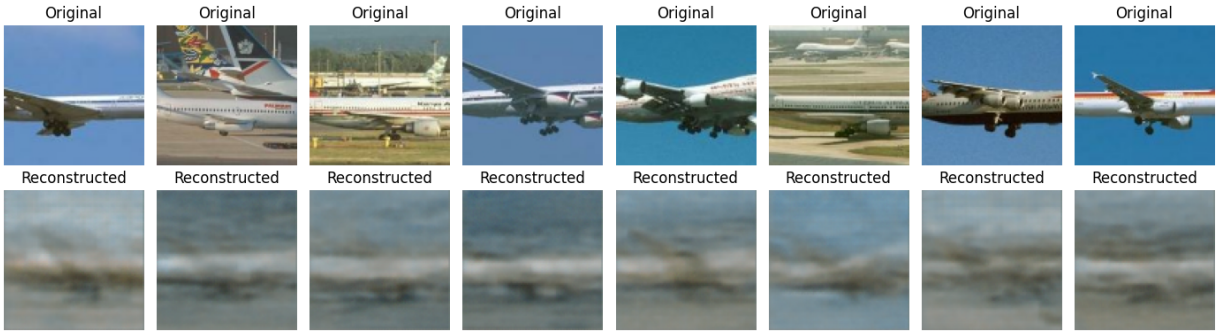
```
                layers.Conv2DTranspose(128, 3, activation="relu", strides=2,␣
↪padding="same", kernel_initializer=tf.keras.initializers.
↪GlorotUniform(seed=RANDOM_SEED)),
                layers.Conv2DTranspose(64, 3, activation="relu", strides=2,␣
↪padding="same", kernel_initializer=tf.keras.initializers.
↪GlorotUniform(seed=RANDOM_SEED)),
                layers.Conv2DTranspose(32, 3, activation="relu", strides=2,␣
↪padding="same", kernel_initializer=tf.keras.initializers.
↪GlorotUniform(seed=RANDOM_SEED)),
                layers.Conv2D(c, 3, activation="sigmoid", padding="same",␣
↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)),
            ]
        )

    def encode(self, x):
        mean_log_var = self.encoder(x)
        mean, log_var = tf.split(mean_log_var, num_or_size_splits=2, axis=1)
        return mean, log_var

    def reparameterize(self, mean, log_var):
        eps = tf.random.normal(shape=tf.shape(mean))
        return mean + eps * tf.exp(log_var * 0.5)

    def decode(self, z):
        return self.decoder(z)

    def call(self, x):
        mean, log_var = self.encode(x)
        z = self.reparameterize(mean, log_var)
        reconstructed = self.decode(z)

        # Add KL divergence loss as a model metric
        kl_loss = -0.5 * tf.reduce_mean(
            tf.reduce_sum(1 + log_var - tf.square(mean) - tf.exp(log_var), axis=1)
        )
        self.add_loss(kl_loss)

        return reconstructed

# Main training function
def train_vae(dataset, img_shape=(224, 224, 3), latent_dim=64, epochs=20,␣
↪learning_rate=0.001):
    # Create VAE
    vae = VAE(img_shape, latent_dim)

    vae.compile(optimizer=keras.optimizers.Adam(learning_rate=learning_rate),␣
↪loss="mse", metrics=["mse"])

    # Train the model
    history = vae.fit(dataset, dataset, batch_size=BATCH_SIZE, epochs=epochs,␣
↪verbose=VERBOSE)
```

```
    return vae, history

# Generate reconstructions from the VAE
def generate_vae_reconstructions(vae, test_images, n=8):
    # Get test images
    test_sample = test_images[:n]
    reconstructed = vae.predict(test_sample, verbose=VERBOSE)
    return test_sample, reconstructed

def visualize_vae_reconstructions(originals, reconstructions, n=8):
    plt.figure(figsize=(14, 4))
    for i in range(min(n, len(originals))):
        # Original
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(originals[i])
        plt.title("Original")
        plt.axis("off")

        # Reconstruction
        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(reconstructions[i])
        plt.title("Reconstructed")
        plt.axis("off")

    plt.tight_layout()
    plt.show()

def combine_metrics(*args, metric_names=None):
    all_metrics = {}

    # Process each experiment's metrics
    for _, metrics_dict in enumerate(args):
        metric_data = metrics_dict[1:]  # Skip header row

        # Initialize dictionary on first run
        if not all_metrics:
            all_metrics = {row[0]: [] for row in metric_data}

        # Add values for this experiment
        for row in metric_data:
            all_metrics[row[0]].append(row[1])


    # Build the table
    header_row = ["Metric"] + [i for i in metric_names]
    combined_table = [header_row]

    # Add each metric row
    for metric, values in all_metrics.items():
        combined_table.append([metric] + values)
```

```
    return combined_table
```

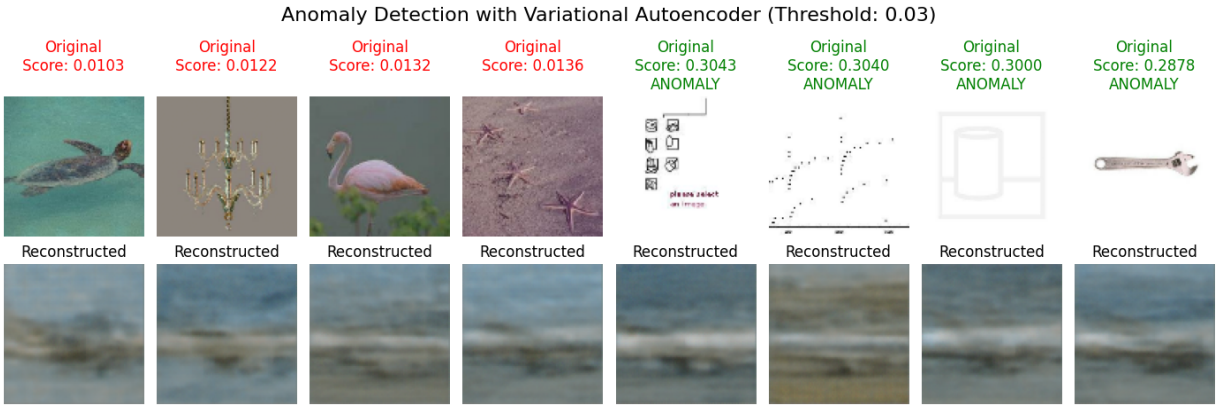
Train the VAE model

[14]:
```
vae, history = train_vae(normal_images, img_shape=(IMAGE_SIZE, IMAGE_SIZE, 3),␣
 ↪latent_dim=LATENT_DIM, epochs=20)

original, reconstructed = generate_vae_reconstructions(vae, normal_images, n=8)
visualize_vae_reconstructions(original, reconstructed, n=8)
```

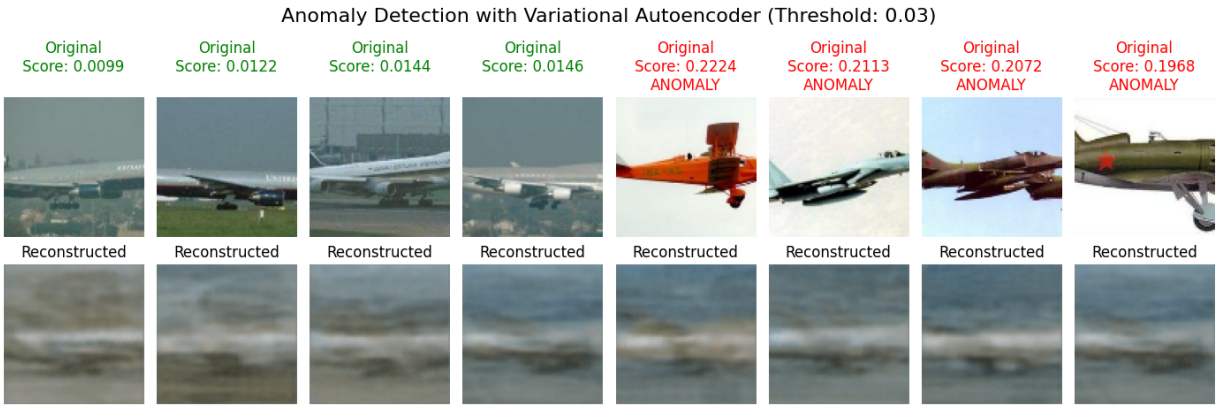


Visualize result of VAE

[15]:
```
def detect_anomalies_vae(vae, images, threshold):
    reconstructed = vae.predict(images, verbose=VERBOSE)

    # Calculate MSE for each image
    mse = np.mean(np.square(images - reconstructed), axis=(1, 2, 3))

    # Determine if each image is an anomaly based on threshold
    anomalies = mse > threshold
    return mse, anomalies, reconstructed

# Detect anomalies with VAE
vae_scores, vae_anomalies, vae_reconstructions = detect_anomalies_vae(vae,␣
 ↪anomaly_images, threshold=THRESHOLD)
visualize_anomalies(anomaly_images, vae_reconstructions, vae_scores, vae_anomalies,␣
 ↪"Variational Autoencoder", is_normal_class=False)
print(f"VAE detected {np.sum(vae_anomalies)} anomalies from {len(anomaly_images)}␣
 ↪images")

normal_scores, normal_anomalies, normal_reconstructions = detect_anomalies_vae(vae,␣
 ↪normal_images, threshold=THRESHOLD)
visualize_anomalies(normal_images, normal_reconstructions, normal_scores,␣
 ↪normal_anomalies, "Variational Autoencoder", is_normal_class=True)
print(f"VAE disclassified {np.sum(normal_anomalies)} normal images from␣
 ↪{len(normal_images)} images")
```




```
VAE detected 8203 anomalies from 8344 images
```




```
VAE disclassified 687 normal images from 800 images
```

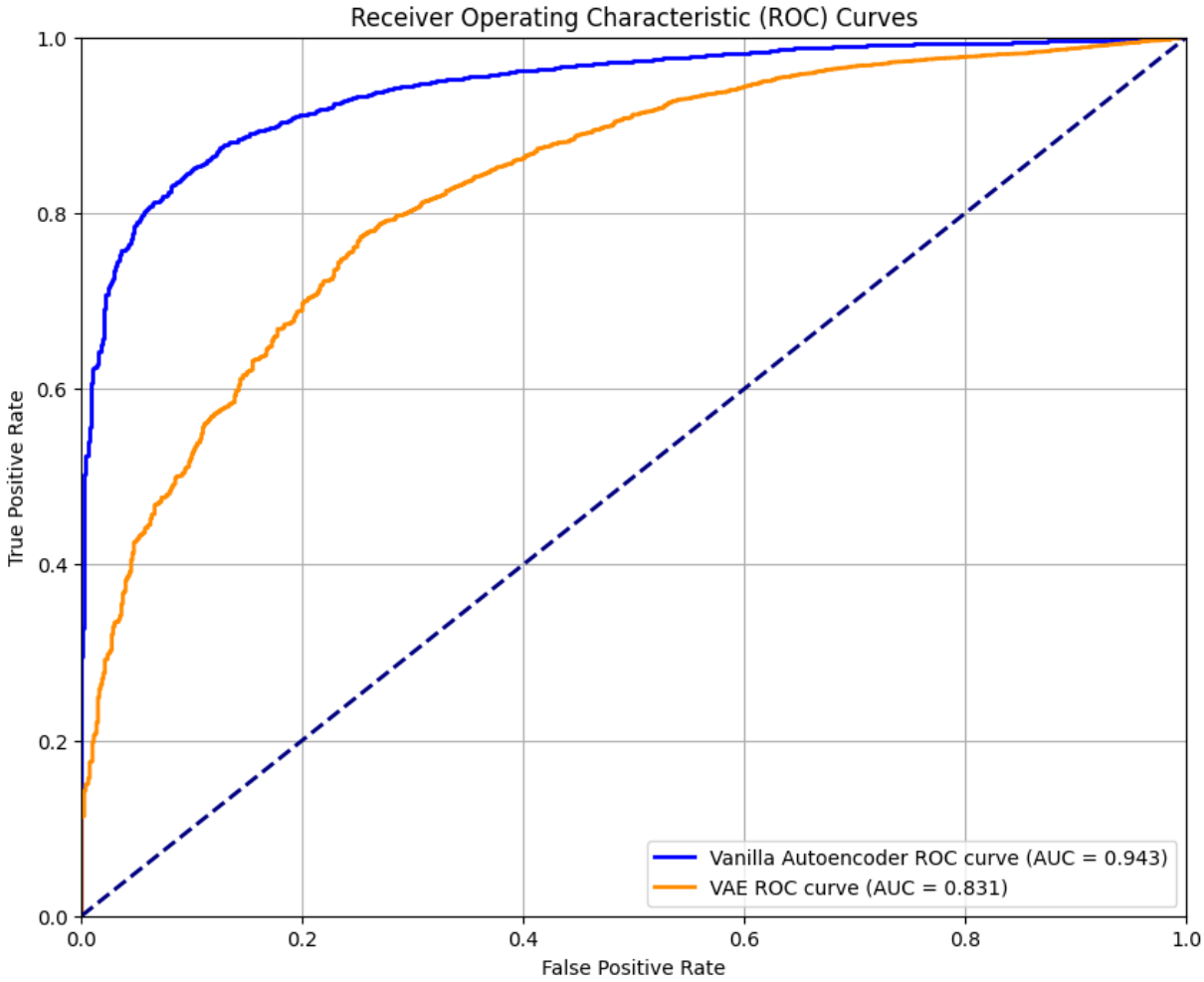
Plot performance metrics of VAE

- Bases on the performance metrics, VAE perform worse than Vanilla Autoencoder in anomaly detection on airplane class in this dataset.

[16]:
```
vae_scores, _, _ = detect_anomalies_vae(vae, test_X, THRESHOLD)
vae_metrics = calculate_metrics(test_y, vae_scores, THRESHOLD)

print("\nAE vs. VAE Performance Metrics:")
combined = combine_metrics(ae_metrics, vae_metrics, metric_names=['AE', 'VAE'])
print(tabulate(combined, headers="firstrow", tablefmt="grid", numalign="right"))

# Calculate ROC AUC score
fpr_vae, tpr_vae, auc_vae = calculate_roc_auc(test_y, vae_scores)

# Plot ROC curves
plt.figure(figsize=(10, 8))
plt.plot(fpr_ae, tpr_ae, color='blue', lw=2, label=f'Vanilla Autoencoder ROC curve␣
↪(AUC = {auc_ae:.3f})')
```

```
plt.plot(fpr_vae, tpr_vae, color='darkorange', lw=2, label=f'VAE ROC curve (AUC =␣
↪{auc_vae:.3f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curves')
plt.legend(loc="lower right")
plt.grid(True)
plt.show()
```

```
AE vs. VAE Performance Metrics:
+--------------------+--------+--------+
| Metric             |     AE |    VAE |
+====================+========+========+
| Accuracy           | 0.8081 | 0.9091 |
+--------------------+--------+--------+
| Precision          | 0.9936 | 0.9224 |
+--------------------+--------+--------+
| Recall             | 0.7948 | 0.9831 |
+--------------------+--------+--------+
| F1 Score           | 0.8831 | 0.9518 |
+--------------------+--------+--------+
|   True Positives   |   6632 |   8203 |
+--------------------+--------+--------+
|   False Positives  |     43 |    690 |
+--------------------+--------+--------+
|   True Negatives   |    757 |    110 |
+--------------------+--------+--------+
|   False Negatives  |   1712 |    141 |
+--------------------+--------+--------+
```



5. -VAE

As we can see, VAE is unsuccesful at reconstructing the pictures. The output are grey images and it's impossible to recognize the planes, anomalies or not.
A reason for that might be that the KL divergence overpowering the reconstruction loss and making the images like that, to deal with this we can add a (beta) term to the KL divergence to reduce it's impact on the function.

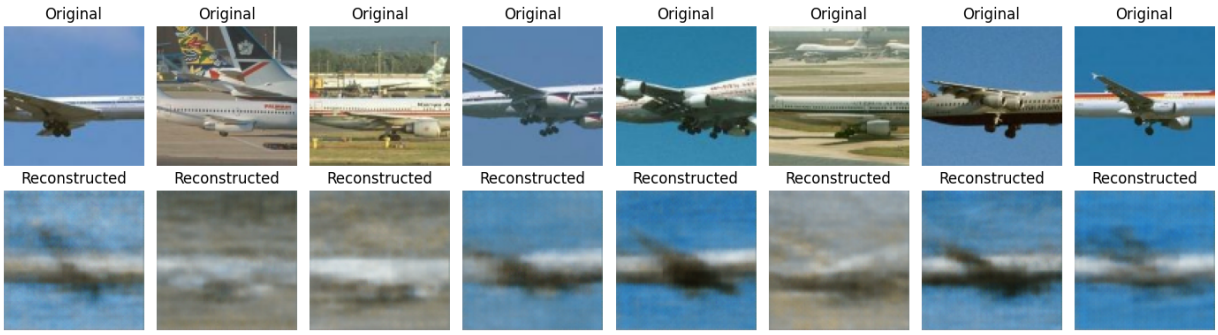
[17]:
```
# Build the VAE model
class VAE(keras.Model):
    def __init__(self, img_shape, latent_dim, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = keras.Sequential(
            [
                layers.Conv2D(32, 3, activation="relu", strides=2, padding="same",␣
↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)),
                layers.Conv2D(64, 3, activation="relu", strides=2, padding="same",␣
↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)),
                layers.Conv2D(128, 3, activation="relu", strides=2, padding="same",␣
↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)),
```

```
            layers.Flatten(),
            layers.Dense(256, activation="relu", kernel_initializer=tf.keras.
↪initializers.GlorotUniform(seed=RANDOM_SEED)),
            layers.Dense(latent_dim * 2, name="latent_vector",␣
↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)),
        ]
    )

    h, w, c = img_shape
    encoder_conv_layers = 3
    decoder_starting_dims = (h // (2 ** encoder_conv_layers), w // (2 **␣
↪encoder_conv_layers), 128)

    self.decoder = keras.Sequential(
        [
            layers.Dense(np.prod(decoder_starting_dims), activation="relu",␣
↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)),
            layers.Reshape(decoder_starting_dims),
            layers.Conv2DTranspose(128, 3, activation="relu", strides=2,␣
↪padding="same", kernel_initializer=tf.keras.initializers.
↪GlorotUniform(seed=RANDOM_SEED)),
            layers.Conv2DTranspose(64, 3, activation="relu", strides=2,␣
↪padding="same", kernel_initializer=tf.keras.initializers.
↪GlorotUniform(seed=RANDOM_SEED)),
            layers.Conv2DTranspose(32, 3, activation="relu", strides=2,␣
↪padding="same", kernel_initializer=tf.keras.initializers.
↪GlorotUniform(seed=RANDOM_SEED)),
            layers.Conv2D(c, 3, activation="sigmoid", padding="same",␣
↪kernel_initializer=tf.keras.initializers.GlorotUniform(seed=RANDOM_SEED)),
        ]
    )

  def encode(self, x):
      mean_log_var = self.encoder(x)
      mean, log_var = tf.split(mean_log_var, num_or_size_splits=2, axis=1)
      return mean, log_var

  def reparameterize(self, mean, log_var):
      eps = tf.random.normal(shape=tf.shape(mean))
      return mean + eps * tf.exp(log_var * 0.5)

  def decode(self, z):
      return self.decoder(z)

  def call(self, x):
      mean, log_var = self.encode(x)
      z = self.reparameterize(mean, log_var)
      reconstructed = self.decode(z)

      # Add KL divergence loss as a model metric
      beta = 0.0001  # add beta
```

```
        kl_loss = beta * (-0.5 * tf.reduce_mean(
            tf.reduce_sum(1 + log_var - tf.square(mean) - tf.exp(log_var), axis=1)
            ))
        self.add_loss(kl_loss)

        return reconstructed

# Main training function
def train_vae(dataset, img_shape=(224, 224, 3), latent_dim=64, epochs=20,␣
 ↪learning_rate=0.001):
    # Create VAE
    vae = VAE(img_shape, latent_dim)
    vae.compile(optimizer=keras.optimizers.Adam(learning_rate=learning_rate),␣
 ↪loss="mse", metrics=["mse"])

    # Train the model
    history = vae.fit(dataset, dataset, batch_size=BATCH_SIZE, epochs=epochs,␣
 ↪verbose=VERBOSE)

    return vae, history

# Generate reconstructions from the VAE
def generate_vae_reconstructions(vae, test_images, n=8):
    # Get test images
    test_sample = test_images[:n]

    reconstructed = vae.predict(test_sample, verbose=VERBOSE)

    return test_sample, reconstructed

def visualize_vae_reconstructions(originals, reconstructions, n=8):
    plt.figure(figsize=(14, 4))
    for i in range(min(n, len(originals))):
        # Original
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(originals[i])
        plt.title("Original")
        plt.axis("off")

        # Reconstruction
        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(reconstructions[i])
        plt.title("Reconstructed")
        plt.axis("off")

    plt.tight_layout()
    plt.show()
```

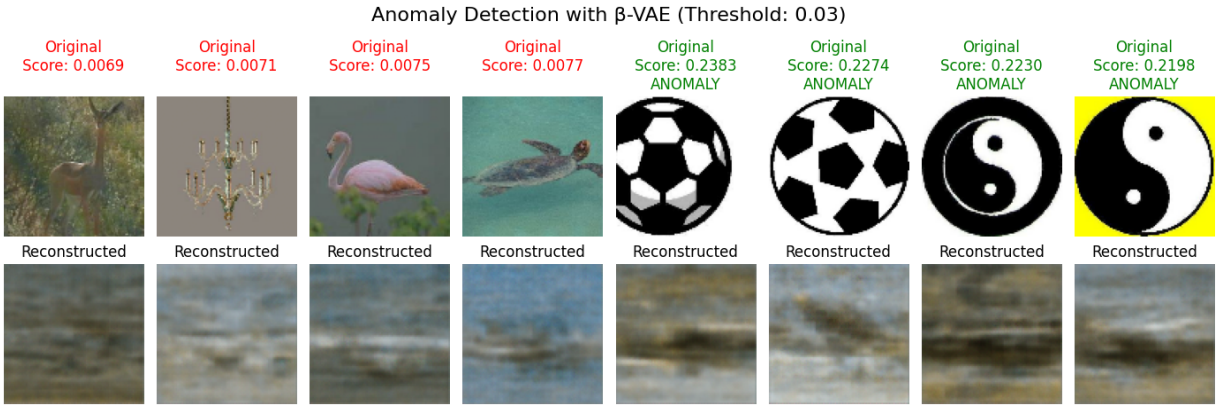
Train the -VAE model

[18]:
```
vae, history = train_vae(normal_images, img_shape=(IMAGE_SIZE, IMAGE_SIZE, 3),␣
↪latent_dim=LATENT_DIM, epochs=20)

original, reconstructed = generate_vae_reconstructions(vae, normal_images, n=8)
visualize_vae_reconstructions(original, reconstructed, n=8)
```

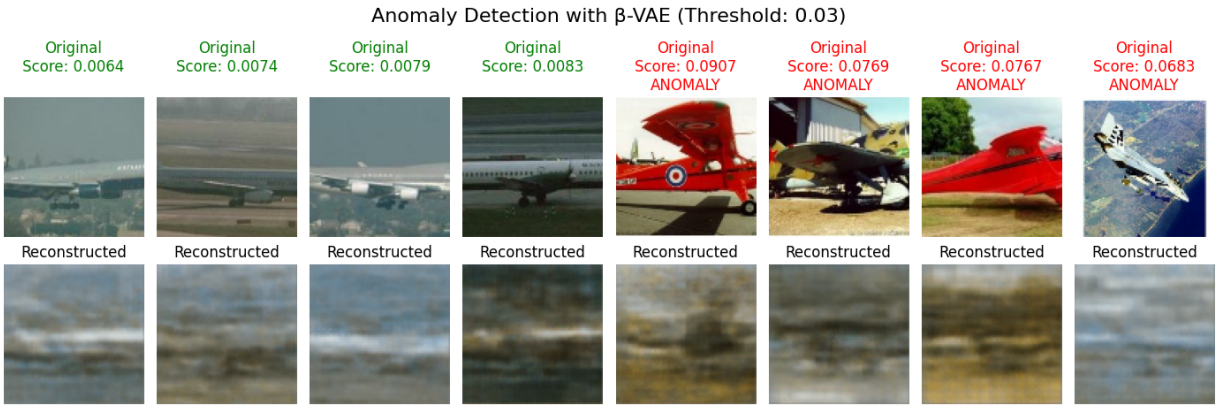


Visualize result of -VAE

[19]:
```
def detect_anomalies_vae(vae, images, threshold):
    reconstructed = vae.predict(images, verbose=VERBOSE)
    # Calculate MSE for each image
    mse = np.mean(np.square(images - reconstructed), axis=(1, 2, 3))
    # Determine if each image is an anomaly based on threshold
    anomalies = mse > threshold

    return mse, anomalies, reconstructed

# Detect anomalies with VAE
vae_scores, vae_anomalies, vae_reconstructions = detect_anomalies_vae(vae,␣
↪anomaly_images, threshold=THRESHOLD)
visualize_anomalies(anomaly_images, vae_reconstructions, vae_scores, vae_anomalies,␣
↪" -VAE", is_normal_class=False)
print(f" -VAE detected {np.sum(vae_anomalies)} anomalies from {len(anomaly_images)}␣
↪images")

normal_scores, normal_anomalies, normal_reconstructions = detect_anomalies_vae(vae,␣
↪normal_images, threshold=THRESHOLD)
visualize_anomalies(normal_images, normal_reconstructions, normal_scores,␣
↪normal_anomalies, " -VAE", is_normal_class=True)
print(f" -VAE disclassified {np.sum(normal_anomalies)} normal images from␣
↪{len(normal_images)} images")
```



-VAE detected 7300 anomalies from 8344 images



-VAE disclassified 154 normal images from 800 images

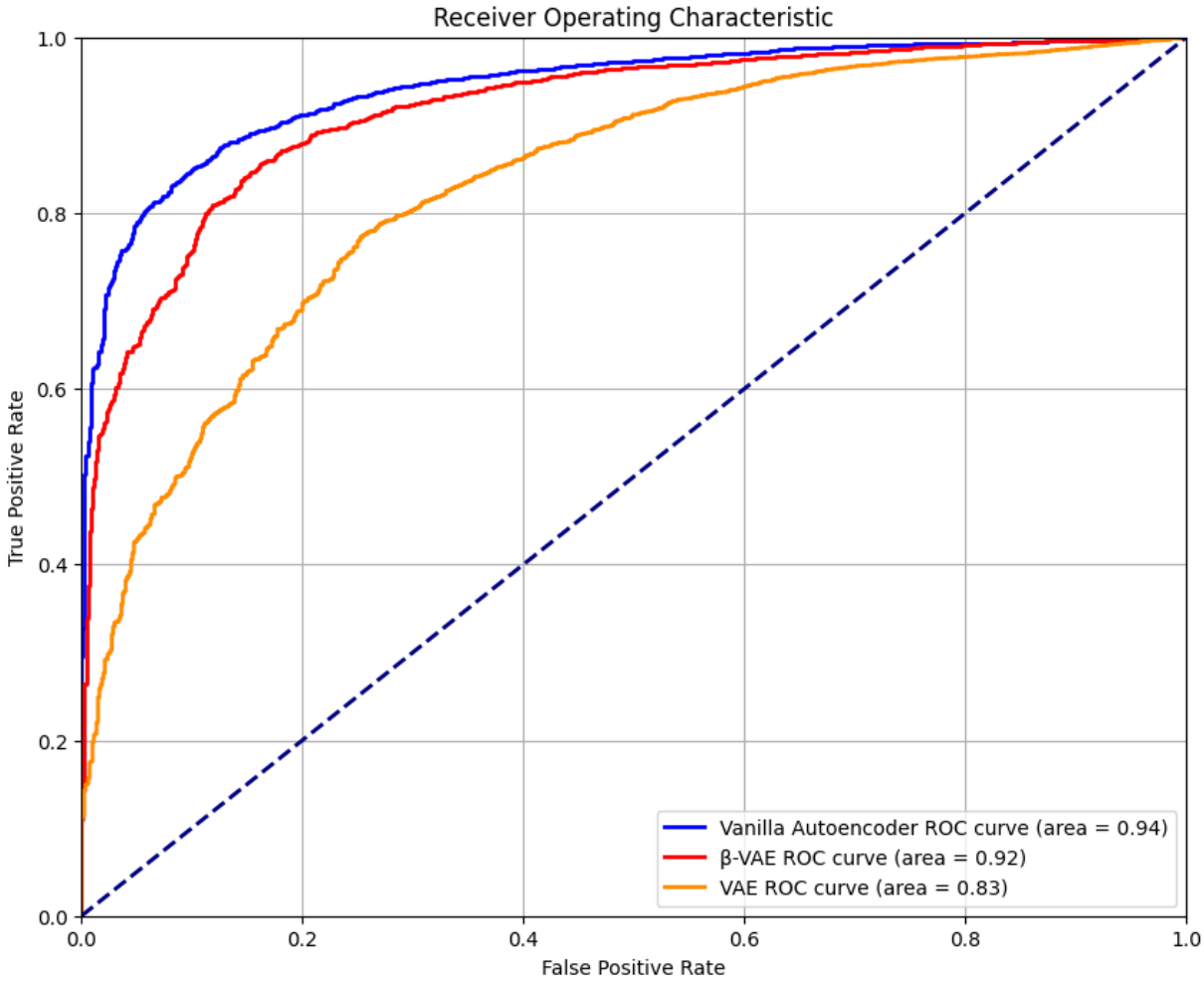
Plot performance metrics of -VAE

[20]:
```
bvae_scores, _, _ = detect_anomalies_vae(vae, test_X, threshold=THRESHOLD)
bvae_metrics = calculate_metrics(test_y, bvae_scores, THRESHOLD)

combined = combine_metrics(ae_metrics, vae_metrics, bvae_metrics, metric_names=['AE',␣
↪'VAE', ' -VAE'])
# Display metrics in a table format
print("\nAE vs. VAE vs.  -VAE Performance Metrics:")
print(tabulate(combined, headers="firstrow", tablefmt="grid", numalign="right"))

# Calculate ROC AUC score
fpr_bvae, tpr_bvae, thresholds  = roc_curve(test_y, bvae_scores)

# Calculate Area Under Curve (AUC)
auc_bvae = auc(fpr_bvae, tpr_bvae)

# Plot ROC curve
plt.figure(figsize=(10, 8))
```

```
plt.plot(fpr_ae, tpr_ae, color='blue', lw=2, label=f'Vanilla Autoencoder ROC curve␣
 ↪(area = {auc_ae:.2f})')
plt.plot(fpr_bvae, tpr_bvae, color='red', lw=2, label=f' -VAE ROC curve (area =␣
 ↪{auc_bvae:.2f})')
plt.plot(fpr_vae, tpr_vae, color='darkorange', lw=2, label=f'VAE ROC curve (area =␣
 ↪{auc_vae:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.grid(True)
plt.show()
```

AE vs. VAE vs. -VAE Performance Metrics:

Metric	AE	VAE	-VAE
Accuracy	0.8081	0.9091	0.8694
Precision	0.9936	0.9224	0.9796
Recall	0.7948	0.9831	0.8751
F1 Score	0.8831	0.9518	0.9244
True Positives	6632	8203	7302
False Positives	43	690	152
True Negatives	757	110	648
False Negatives	1712	141	1042



Conclusion

1. Vanilla Autoencoder is the best model for anomaly detection on airplane class in this dataset.
2. Best setting for anomaly detection on this dataset is:

- Vanilla Autoencoder with threshold `0.03` and latent dimension `1280` and `5` hidden layers for balanced performance.
- Vanilla Autoencoder with threshold `0.04` and latent dimension `1024` and `3` hidden layers for real life scenario (Maximize False Positive Rate).

3. Alternative model performance:

- Vanilla Autoencoder $>$ -VAE $>$ VAE