GENERATIVE AI
ASSIGNMENT # 2
REPORT

Name: Alaiba Nawaz
Roll Number: 21L-5650
Section: BDS-8A

Question 1

```
# Function to check for valid images
def is_valid(example):
    return example["cropped_image"] is not None

filtered_dataset = dataset["test"].filter(is_valid)

print(f"Original dataset size: {len(dataset['test'])}")
print(f"Filtered dataset size: {len(filtered_dataset)}")
```

```
Filter:   0%|          | 0/67170 [00:00<?, ? examples/s]
Original dataset size: 67170
Filtered dataset size: 66787
```

This function makes sure that only valid images stays in the dataset that is good for model.

```
images = filtered_dataset['cropped_image']
labels = filtered_dataset['labels']
```

Real label is 0 and Spoof is 1

```
train_images, test_images, train_labels, test_labels = train_test_split( images, labels, test_size=0.05, random_state=42)
print(f"Train Set: {len(train_images)} images")
print(f"Test Set: {len(test_images)} images")
```

```
Train Set: 63447 images
Test Set: 3340 images
```

Train Set: 63447 images
Test Set: 3340 images

```python
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])

class FaceSpoofDataset(torch.utils.data.Dataset):
    def __init__(self, images, labels, transform=None):
        self.images = images
        self.labels = labels
        self.transform = transform

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        if isinstance(image, str):
            image = Image.open(image).convert("RGB")

        if self.transform:
            image = self.transform(image)

        return image, label

    def __len__(self):
        return len(self.images)
```

We transform the images as follows:

- **Resizes images to** 224x224 pixels.
- **Converts them** into PyTorch tensors.
- **Normalizes pixel values** to have a mean of 0.5 and a standard deviation of 0.5 for all three RGB channels

Resizing and normalizing images helps ensure consistency for the model.
Using a PyTorch dataset makes it easier to load and preprocess images dynamically during training.

```python
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=2e-5)
# Training function
def train_model(model, train_loader, criterion, optimizer, num_epochs=5):
    model.train()
    history = {'train_loss': [], 'train_acc': []}
    for epoch in range(num_epochs):
        running_loss = 0.0
        correct = 0
        total = 0
        for i, (inputs, labels) in enumerate(train_loader):
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()# Zero the parameter gradients

            outputs = model(pixel_values=inputs).logits  # Forward pass
            loss = criterion(outputs, labels)

            loss.backward()# Backward pass and optimize
            optimizer.step()

            running_loss += loss.item() X# Statistics
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if (i+1) % 100 == 0:
                print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{len(train_loader)}], Loss: {loss.item():.4f}')
        epoch_loss = running_loss / len(train_loader)
        epoch_acc = 100 * correct / total
        history['train_loss'].append(epoch_loss)
        history['train_acc'].append(epoch_acc)
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}, Accuracy: {epoch_acc:.2f}%')
    return history
```

This train function trains a deep learning model using .
CrossEntropyLoss: Defines the loss function as cross-entropy loss, which is commonly used for classification tasks.

AdamW optimizer: Uses the AdamW optimizer (a variant of Adam with weight decay that penalizes large weights in a neural network by adding a scaled L2 norm of the weights to the loss function. This prevents overfitting by encouraging smaller, more generalizable weights.) to update model parameters. The learning rate is set to 2e-5 (0.00002), which is small to ensure stable training.

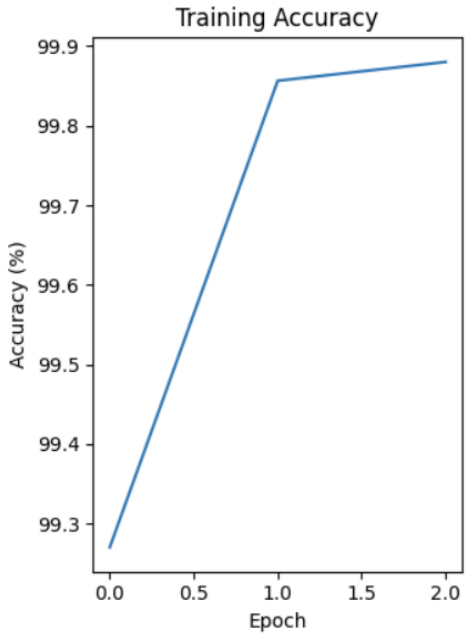
Loop over each epoch (num_epochs times).
running_loss = 0.0 → Accumulates loss over batches for averaging later.
correct = 0 & total = 0 → Tracks the number of correct predictions for accuracy calculation.

Loop Over Training Batches

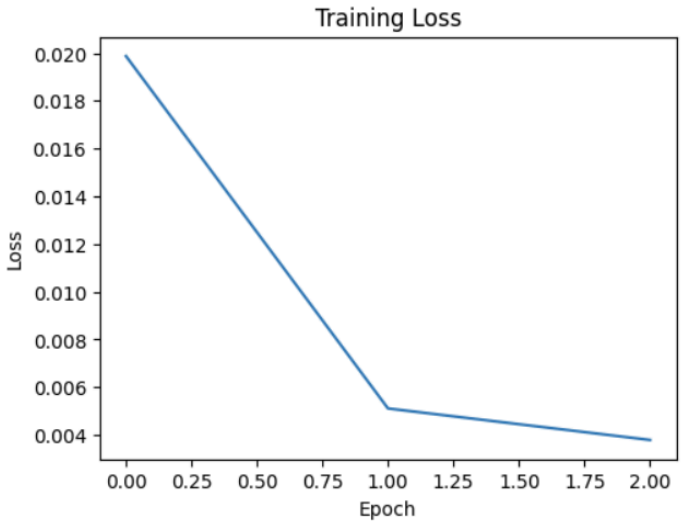
- Iterate over batches of (inputs, labels) from train_loader.
- Move data to GPU (device) if available for faster computation.
- **optimizer.zero_grad()** → Clears previous gradients to prevent accumulation (important in PyTorch).
- **outputs = model(pixel_values=inputs).logits** → The model makes predictions (logits) on the input data.
- **loss = criterion(outputs, labels)** → Computes the loss between predictions and actual labels.
- **loss.backward()** → Computes gradients via backpropagation.
- **optimizer.step()** → Updates the model's parameters using the computed gradients.

- **running_loss += loss.item()** → Adds the batch loss to running_loss for averaging.
- **_, predicted = torch.max(outputs.data, 1)** → Gets the predicted class with the highest probability.
- **total += labels.size(0)** → Increases the total count of processed samples.
- **correct += (predicted == labels).sum().item()** → Counts the number of correctly predicted samples.
- **epoch_loss = running_loss / len(train_loader)** → Computes the average loss for the epoch.
- **epoch_acc = 100 * correct / total** → Computes accuracy percentage.
- Store loss and accuracy in history for later analysis.

It loops through multiple epochs, performing forward propagation, loss computation, backpropagation, and weight updates for each batch of training data. The function tracks training loss and accuracy, prints progress every 100 steps, and returns a history dictionary for analysis.



Training accuracy increases over epochs which means classification performance is better.



Training loss decreases over epochs which means the model is learning and improving.

Accuracy (99.76%): The model correctly classifies 99.76% of the total samples.
Precision (99.66%): Out of all predicted positives, 99.66% are actually correct.
Recall (100%): The model identifies all actual positive cases without missing any.
F1 Score (99.83%): A balanced measure of precision and recall, showing strong overall performance.
Confusion Matrix: The model misclassified 8 negative samples as positive but correctly classified all positive samples.

Prediction: Real

Prediction: Spoof

Question 2

Downloaded the datasets.

```
def load_clip():
    model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
    processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
    return model, processor
```

Loads the CLIP model and processor from OpenAI's pretrained "clip-vit-base-patch32" model.

```
# Get a sample of images to encode
def get_image_paths(num_images=100):
    image_dir = "coco/val2017"
    all_images = [os.path.join(image_dir, f) for f in os.listdir(image_dir) if f.endswith('.jpg')]
    return random.sample(all_images, min(num_images, len(all_images)))
```

Retrieves a random sample of image file paths from the COCO validation dataset (val2017).
Filters only .jpg images and randomly selects num_images from the dataset.Instead of processing the entire dataset, we sample a subset to make computations efficient and manageable.

```
def encode_images(model, processor, image_paths):
    image_features = []

    for img_path in image_paths:
        image = Image.open(img_path).convert("RGB")
        inputs = processor(images=image, return_tensors="pt")

        with torch.no_grad():
            features = model.get_image_features(**inputs)
            features = features / features.norm(dim=1, keepdim=True)
            image_features.append(features)
```

Converts images into feature vectors (embeddings) using CLIP.
Each image is processed using CLIP Processor, passed through the model, and normalized to ensure consistency in comparisons.Image embeddings are required to compute similarity with text embeddings later.

```python
def find_similar_images(model, processor, image_features, image_paths, text_query, top_k=5):
    # Process text
    inputs = processor(text=text_query, return_tensors="pt")

    with torch.no_grad():
        text_features = model.get_text_features(**inputs)
        text_features = text_features / text_features.norm(dim=1, keepdim=True)  # normalize

    # Calculate similarity
    similarity = (100.0 * text_features @ image_features.T).softmax(dim=-1)
    # Get top results
    values, indices = similarity[0].topk(top_k)

    top_images = [image_paths[i] for i in indices]
    top_scores = values.tolist()

    return top_images, top_scores
```

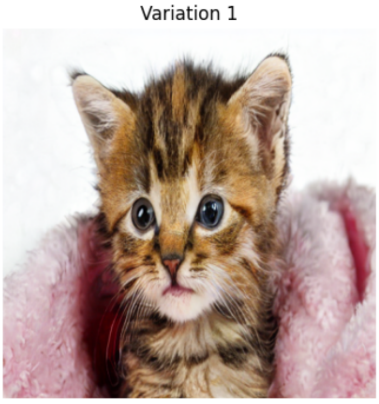
Converts a text query into an embedding using CLIP.
Computes the similarity between the text embedding and image embeddings using dot product and applies softmax for ranking.
Retrieves the top k most relevant images based on similarity scores.
This allows us to find images that match a given text description, enabling text-to-image retrieval.

RESULTS:

QUESTION 3

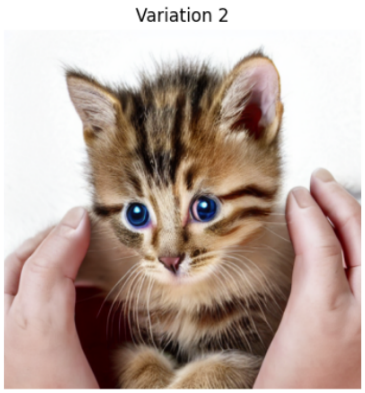
Original Image

Variation 1 (strength: 0.2, guidance_scale: 7, num_inference_steps: 25)



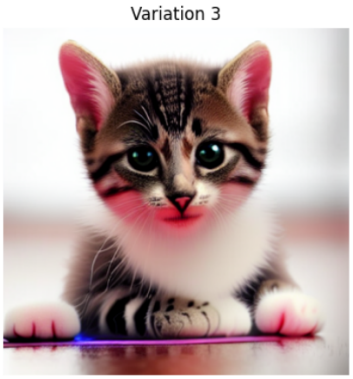
Detail: It's almost the same as the original but colors are a bit dark .
Realism: Very realistic.
Coherence: Strong coherence with the original image

Variation 2 (strength: 0.6, guidance_scale: 10, num_inference_steps: 50)



Detail:It has many changes, with slight changes in fur texture and facial features also it made the blanket appear as hands.
Realism: Looks like AI generated and the eyes are blue now .
Coherence: Some deviation from the original, but the image still resembles the input closely.

Variation 3 (strength: 0.9, guidance_scale: 12, num_inference_steps: 100)



- **Detail**: Everything is changed now shape, fur, and lighting, making it look like a different cat.
- **Realism**: The image looks more fake.
- **Coherence**: The least coherent with the original image, as it introduces creative distortions.

Increasing strength leads to more transformation.

Higher guidance_scale enforces a stronger AI interpretation.

More inference_steps improve refinement but may also amplify unrealistic details.

Watercolor cat – Looks like a soft and colorful painting.

Pixel art cat – Too real, doesn't look like pixel art.

Surreal Dalí cat – A little dreamy but not super weird like Dalí's art.

Van Gogh cat – Bright colors and brush strokes match Van Gogh's style well.

Cyberpunk robotic cat – Glowing eyes and metal parts make it look futuristic.