input_transformation

April 11, 2025

[ ]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input
```

[ ]:
```
# Hyperparameters
BATCH_SIZE = 64
IMG_SIZE = 96  # Upscale CIFAR-10 images (32x32) to 96x96 for MobileNetV2
AUTOTUNE = tf.data.AUTOTUNE
```

[ ]:
```
def resize_and_preprocess(image, label):
    image = tf.cast(image, tf.float32)
    image = tf.image.resize(image, [IMG_SIZE, IMG_SIZE])
    image = preprocess_input(image)
    return image, label
```

[ ]:
```
# Load CIFAR-10 test dataset
(_, _), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
y_test = np.squeeze(y_test)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071          13s
0us/step
```

[ ]:
```
model = tf.keras.models.load_model("model.keras")
```

[ ]:
```
#preprocessing data
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_dataset = test_dataset.map(resize_and_preprocess,␣
 ↪num_parallel_calls=AUTOTUNE)
test_dataset = test_dataset.batch(BATCH_SIZE).prefetch(AUTOTUNE)
```

[ ]:
```
loss, accuracy = model.evaluate(test_dataset)
```

```
157/157             13s 32ms/step -
accuracy: 0.9132 - loss: 0.2790
```

[ ]:
```
loss, accuracy
```

[ ]: (0.2672818899154663, 0.9153000116348267)

[ ]:
```
@tf.function
def batched_fgsm_attack(images, labels, epsilon=0.01):
    with tf.GradientTape() as tape:
        tape.watch(images)
        predictions = model(images, training=False)
        loss = tf.keras.losses.sparse_categorical_crossentropy(labels,␣
↪predictions)
    gradients = tape.gradient(loss, images)
    adv_images = images + epsilon * tf.sign(gradients)
    adv_images = tf.clip_by_value(adv_images, -1, 1)
    return adv_images
```

[ ]:
```
@tf.function
def batched_pgd_attack(images, labels, epsilon=0.01, alpha=0.005, num_iter=10):
    adv_images = tf.identity(images)

    for _ in tf.range(num_iter):
        with tf.GradientTape() as tape:
            tape.watch(adv_images)
            predictions = model(adv_images, training=False)
            loss = tf.keras.losses.sparse_categorical_crossentropy(labels,␣
↪predictions)
        gradients = tape.gradient(loss, adv_images)
        adv_images = adv_images + alpha * tf.sign(gradients)

        # Project perturbation
        perturbation = tf.clip_by_value(adv_images - images, -epsilon, epsilon)
        adv_images = tf.clip_by_value(images + perturbation, -1, 1)

    return adv_images
```

[ ]:
```
def deepfool_attack(image, num_classes=10, overshoot=0.0000001, max_iter=1):
    image = tf.convert_to_tensor(image, dtype=tf.float32)
    perturbed_image = tf.identity(image)

    # Get original prediction and label
    with tf.GradientTape() as tape:
        tape.watch(perturbed_image)
        logits = model(tf.expand_dims(perturbed_image, axis=0))[0]
    orig_label = tf.argmax(logits)

    r_tot = tf.zeros_like(image)
    i = 0

    while i < max_iter:
```

```
        with tf.GradientTape(persistent=True) as tape:
            tape.watch(perturbed_image)
            logits = model(tf.expand_dims(perturbed_image, axis=0))[0]

        current_label = tf.argmax(logits)
        if current_label != orig_label:
            break

        # Compute gradients for all class logits
        gradients = []
        for k in range(num_classes):
            with tf.GradientTape() as tape2:
                tape2.watch(perturbed_image)
                logit_k = model(tf.expand_dims(perturbed_image, axis=0))[0, k]
            grad_k = tape2.gradient(logit_k, perturbed_image)
            gradients.append(grad_k)
        gradients = tf.stack(gradients)

        # Compute minimal perturbation
        f_orig = logits[orig_label]
        perturbs = []
        for k in range(num_classes):
            if k == orig_label:
                continue
            w_k = gradients[k] - gradients[orig_label]
            f_k = logits[k] - f_orig
            norm_w = tf.norm(tf.reshape(w_k, [-1])) + 1e-8
            pert_k = tf.abs(f_k) / norm_w
            perturbs.append((pert_k, w_k))

        # Choose the closest decision boundary
        perturbs.sort(key=lambda x: x[0])
        pert_k, w_k = perturbs[0]

        # Compute minimal directional perturbation (no sign scaling)
        r_i = (pert_k * w_k) / (tf.norm(w_k) + 1e-8)
        r_tot += r_i

        # Apply accumulated perturbation with small overshoot
        perturbed_image = image + (1 + overshoot) * r_tot
        perturbed_image = tf.clip_by_value(perturbed_image, -1, 1)

        i += 1

    return perturbed_image
```

[ ]:
```python
def get_test_dataset():
    # Load CIFAR-10 test dataset and preprocess
    (_, _), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
    y_test = np.squeeze(y_test)
    ds = tf.data.Dataset.from_tensor_slices((x_test, y_test))
    ds = ds.map(resize_and_preprocess, num_parallel_calls=AUTOTUNE)
    ds = ds.batch(BATCH_SIZE).prefetch(AUTOTUNE)
    return ds
```

[ ]:
```python
clean_ds = get_test_dataset()
model.compile(loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

[ ]:
```python
def build_adversarial_dataset_fast(dataset, attack_fn, attack_name="FGSM"):
    adv_images_all = []
    adv_labels_all = []

    print(f"\nBuilding {attack_name} dataset...")

    for images, labels in dataset:
        adv_images = attack_fn(images, labels)
        adv_images_all.append(adv_images)
        adv_labels_all.append(labels)

    adv_images_all = tf.concat(adv_images_all, axis=0)
    adv_labels_all = tf.concat(adv_labels_all, axis=0)

    adv_ds = tf.data.Dataset.from_tensor_slices((adv_images_all,␣
↪adv_labels_all))
    return adv_ds.batch(BATCH_SIZE).prefetch(AUTOTUNE)
```

[ ]:
```python
def build_adversarial_dataset_deepfool(attack_fn, name="DeepFool",␣
↪max_samples=500, num_classes=10):
    adv_images = []
    adv_labels = []

    print(f"\nGenerating {name} adversarial dataset (max {max_samples} samples).
↪..")
    sample_count = 0

    for images, labels in clean_ds:
        for img, label in zip(images, labels):
            # Pass a fixed number of classes instead of the label value.
            adv_img = attack_fn(img, num_classes)
            adv_images.append(adv_img.numpy())
            adv_labels.append(int(label.numpy()))
            sample_count += 1
```

```
            if sample_count >= max_samples:
                break
        if sample_count >= max_samples:
            break

    adv_images = np.array(adv_images)
    adv_labels = np.array(adv_labels)

    ds = tf.data.Dataset.from_tensor_slices((adv_images, adv_labels))
    ds = ds.batch(BATCH_SIZE).prefetch(AUTOTUNE)
    return ds
```

[ ]:
```
def evaluate_model_on_dataset(dataset, name="Dataset"):
    y_true, y_pred = [], []
    total_loss = 0.0
    total_samples = 0

    loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
    for batch_images, batch_labels in dataset:
        preds = model(batch_images, training=False)
        loss = loss_fn(batch_labels, preds).numpy()
        pred_classes = tf.argmax(preds, axis=1).numpy()
        y_true.extend(batch_labels.numpy())
        y_pred.extend(pred_classes)
        total_loss += loss * len(batch_labels)
        total_samples += len(batch_labels)

    accuracy = np.mean(np.array(y_true) == np.array(y_pred))
    avg_loss = total_loss / total_samples
    correct = sum(np.array(y_true) == np.array(y_pred))
    incorrect = total_samples - correct

    print(f"\n{name} Evaluation:")
    print(f"  Total Samples: {total_samples}")
    print(f"  Accuracy: {accuracy:.4f}")
    print(f"  Loss: {avg_loss:.4f}")
    print(f"  Correct Predictions: {correct}")
    print(f"  Incorrect Predictions: {incorrect}")
    return accuracy, avg_loss
```

[ ]:
```
fgsm_ds = build_adversarial_dataset_fast(clean_ds, lambda x, y:␣
 ↪batched_fgsm_attack(x, y, epsilon=0.01), attack_name="FGSM")
pgd_ds = build_adversarial_dataset_fast(clean_ds, lambda x, y:␣
 ↪batched_pgd_attack(x, y, epsilon=0.01, alpha=0.005, num_iter=10),␣
 ↪attack_name="PGD")
```

```
Building FGSM dataset…

Building PGD dataset…
```

[ ]:
```
evaluate_model_on_dataset(fgsm_ds, name="FGSM")
```

```
FGSM Evaluation:
  Total Samples: 10000
  Accuracy: 0.1820
  Loss: 5.1916
  Correct Predictions: 1820
  Incorrect Predictions: 8180
```

[ ]:
```
(np.float64(0.182), np.float32(5.191604))
```

[ ]:
```
evaluate_model_on_dataset(pgd_ds, name="PGD")
```

```
PGD Evaluation:
  Total Samples: 10000
  Accuracy: 0.0000
  Loss: 22.0664
  Correct Predictions: 0
  Incorrect Predictions: 10000
```

[ ]:
```
(np.float64(0.0), np.float32(22.066427))
```

[ ]:
```
deepfool_ds = build_adversarial_dataset_deepfool(deepfool_attack,␣
 ↪name="DeepFool", max_samples=200)
evaluate_model_on_dataset(deepfool_ds, name="DeepFool Attack")
```

```
Generating DeepFool adversarial dataset (max 200 samples)…

DeepFool Attack Evaluation:
  Total Samples: 200
  Accuracy: 0.1500
  Loss: 5.1534
  Correct Predictions: 30
  Incorrect Predictions: 170
```

[ ]:
```
(np.float64(0.15), np.float32(5.153436))
```

[ ]:
```
def get_gaussian_kernel(size=3, sigma=1.0):
    """Creates a 2D Gaussian kernel."""
    x = tf.range(-size // 2 + 1, size // 2 + 1, dtype=tf.float32)
    x = tf.exp(-(x**2) / (2 * sigma**2))
```

```python
    kernel_1d = x / tf.reduce_sum(x)
    kernel_2d = tf.tensordot(kernel_1d, kernel_1d, axes=0)
    kernel_2d = kernel_2d / tf.reduce_sum(kernel_2d)
    return kernel_2d[:, :, tf.newaxis, tf.newaxis]  # Shape: (H, W,␣
↪in_channels=1, out_channels=1)

def apply_gaussian_blur(x, sigma):
    """Applies Gaussian blur using depthwise convolution."""
    kernel = get_gaussian_kernel(size=3, sigma=sigma)
    channels = tf.shape(x)[-1]
    kernel = tf.tile(kernel, [1, 1, channels, 1])  # Make kernel channel-wise
    x = tf.nn.depthwise_conv2d(x, kernel, strides=[1, 1, 1, 1], padding='SAME')
    return x
```

[ ]:
```python
def inference_input_transformation(
    x,
    apply_bitdepth=True,
    bits=4,
    apply_noise=True,
    noise_std=0.05,
    apply_jpeg=True,
    jpeg_quality=75,
    apply_blur=True,
    blur_sigma=0.5
):
    """
    Apply input transformations: quantization, noise, JPEG compression, and␣
↪blur.

    Args:
        x (Tensor): Input tensor in [0,1].
    """
    if apply_bitdepth:
        levels = 2 ** bits
        x = tf.round(x * (levels - 1)) / (levels - 1)

    if apply_noise:
        noise = tf.random.normal(tf.shape(x), mean=0.0, stddev=noise_std,␣
↪dtype=x.dtype)
        x = x + noise

    if apply_jpeg:
        def jpeg_fn(img):
            img_uint8 = tf.image.convert_image_dtype(img, tf.uint8)
            encoded = tf.io.encode_jpeg(img_uint8, quality=jpeg_quality)
            decoded = tf.io.decode_jpeg(encoded)
            return tf.image.convert_image_dtype(decoded, tf.float32)
```

```python
        x = tf.map_fn(jpeg_fn, x)

    if apply_blur:
        x = apply_gaussian_blur(x, sigma=blur_sigma)

    x = tf.clip_by_value(x, 0.0, 1.0)
    return x
```

[ ]:
```python
class TransformedModel(tf.keras.Model):
    def __init__(
        self,
        base_model,
        apply_bitdepth=True,
        bits=4,
        apply_noise=True,
        noise_std=0.05,
        apply_jpeg=True,
        jpeg_quality=75,
        apply_blur=True,
        blur_sigma=0.5
    ):
        super().__init__()
        self.base_model = base_model
        self.apply_bitdepth = apply_bitdepth
        self.bits = bits
        self.apply_noise = apply_noise
        self.noise_std = noise_std
        self.apply_jpeg = apply_jpeg
        self.jpeg_quality = jpeg_quality
        self.apply_blur = apply_blur
        self.blur_sigma = blur_sigma

    def call(self, inputs, training=False):
        # Convert from [-1, 1] to [0, 1] before transformation
        inputs = (inputs + 1.0) / 2.0

        transformed = inference_input_transformation(
            inputs,
            apply_bitdepth=self.apply_bitdepth,
            bits=self.bits,
            apply_noise=self.apply_noise,
            noise_std=self.noise_std,
            apply_jpeg=self.apply_jpeg,
            jpeg_quality=self.jpeg_quality,
            apply_blur=self.apply_blur,
            blur_sigma=self.blur_sigma
        )
```

```
        # Convert back to [-1, 1] for model input
        transformed = transformed * 2.0 - 1.0
        return self.base_model(transformed, training=training)
```

[ ]:
```
model = TransformedModel(model)
```

[ ]:
```
evaluate_model_on_dataset(clean_ds, name='Clean + transformed')
```

```
Clean + transformed Evaluation:
  Total Samples: 10000
  Accuracy: 0.4095
  Loss: 2.4089
  Correct Predictions: 4095
  Incorrect Predictions: 5905
```

[ ]:
```
(np.float64(0.4095), np.float32(2.4088583))
```

[ ]:
```
evaluate_model_on_dataset(fgsm_ds, name="FGSM + Transformed")
```

```
FGSM + Transformed Evaluation:
  Total Samples: 10000
  Accuracy: 0.3898
  Loss: 2.5223
  Correct Predictions: 3898
  Incorrect Predictions: 6102
```

[ ]:
```
(np.float64(0.3898), np.float32(2.5223048))
```

[ ]:
```
evaluate_model_on_dataset(pgd_ds, name="PGD + Transformed")
```

```
PGD + Transformed Evaluation:
  Total Samples: 10000
  Accuracy: 0.3863
  Loss: 2.4916
  Correct Predictions: 3863
  Incorrect Predictions: 6137
```

[ ]:
```
(np.float64(0.3863), np.float32(2.4916496))
```

[ ]:
```
evaluate_model_on_dataset(deepfool_ds, name="DeepFool + Transformed")
```

```
DeepFool + Transformed Evaluation:
  Total Samples: 200
```

```
Accuracy: 0.1900
Loss: 4.7012
Correct Predictions: 38
Incorrect Predictions: 162
```

[ ]: (np.float64(0.19), np.float32(4.7012343))

[ ]: