Introduction to TensorFlow Operations

TensorFlow is a powerful open-source library for numerical computation and machine learning. It uses data flow graphs to represent computation, where nodes represent operations (ops) and edges represent data (tensors) that flow between these operations.

Key Concepts

- **Tensors**: Multi-dimensional arrays used as the basic data structure in TensorFlow.
- **Operations (Ops)**: Functions that take tensors as input and produce tensors as output.
- **Graphs**: A set of computations that are performed to process data.

Creating Tensors

Tensors are the fundamental building blocks in TensorFlow. You can create tensors using various functions provided by TensorFlow.

```
import tensorflow as tf

# Create a constant tensor

tensor_a = tf.constant([[1, 2], [3, 4]])

print("Tensor A:")

print(tensor_a)

# Create a tensor filled with zeros

tensor_b = tf.zeros([2, 3])

print("\nTensor B:")

print(tensor_b)

# Create a tensor with random values

tensor_c = tf.random.uniform([2, 2], minval=0, maxval=10)

print("\nTensor C:")

print(tensor_c)
```

```
Tensor A:
tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)

Tensor B:
tf.Tensor(
[[0. 0. 0.]
 [0. 0. 0.]], shape=(2, 3), dtype=float32)

Tensor C:
tf.Tensor(
[[7.4682546 8.87848   ]
 [2.2592866 6.6578293]], shape=(2, 2), dtype=float32)
```

Mathematical, Reduction, and Matrix operations

Mathematical Operations

TensorFlow provides a wide range of mathematical operations that can be performed on tensors. These operations are essential for building and training machine learning models.

Example: Mathematical Operations

```
import tensorflow as tf


# Define tensors

a = tf.constant([2, 4, 6])

b = tf.constant([1, 3, 5])


# Addition

add = tf.add(a, b)

print("Addition:", add.numpy())


# Subtraction

sub = tf.subtract(a, b)

print("Subtraction:", sub.numpy())


# Multiplication

mul = tf.multiply(a, b)
```

```
print("Multiplication:", mul.numpy())


# Division
div = tf.divide(a, b)
print("Division:", div.numpy())


# Power
power = tf.pow(a, 2)
print("Power:", power.numpy())


# Square root
sqrt = tf.sqrt(tf.cast(a, tf.float32))
print("Square Root:", sqrt.numpy())
```

```
Addition: [ 3  7 11]
Subtraction: [1 1 1]
Multiplication: [ 2 12 30]
Division: [2.         1.33333333 1.2       ]
Power: [ 4 16 36]
Square Root: [1.4142135 2.        2.4494898]
```

Reduction Operations

Reduction operations are used to reduce tensors along certain dimensions. These operations are useful for summarizing information from tensors.

Example: Reduction Operations

```
# Define a 2D tensor
tensor = tf.constant([[1, 2, 3], [4, 5, 6]])


# Reduce sum
reduce_sum = tf.reduce_sum(tensor)
print("Reduce Sum:", reduce_sum.numpy())


# Reduce sum along axis 0 (columns)
reduce_sum_axis0 = tf.reduce_sum(tensor, axis=0)
```

```
print("Reduce Sum along Axis 0:", reduce_sum_axis0.numpy())

# Reduce sum along axis 1 (rows)
reduce_sum_axis1 = tf.reduce_sum(tensor, axis=1)
print("Reduce Sum along Axis 1:", reduce_sum_axis1.numpy())

# Reduce mean
reduce_mean = tf.reduce_mean(tensor)
print("Reduce Mean:", reduce_mean.numpy())

# Reduce mean along axis 0
reduce_mean_axis0 = tf.reduce_mean(tensor, axis=0)
print("Reduce Mean along Axis 0:", reduce_mean_axis0.numpy())

# Reduce mean along axis 1
reduce_mean_axis1 = tf.reduce_mean(tensor, axis=1)
print("Reduce Mean along Axis 1:", reduce_mean_axis1.numpy())

# Reduce max
reduce_max = tf.reduce_max(tensor)
print("Reduce Max:", reduce_max.numpy())

# Reduce min
reduce_min = tf.reduce_min(tensor)
print("Reduce Min:", reduce_min.numpy())
```

```
Reduce Sum: 21
Reduce Sum along Axis 0: [5 7 9]
Reduce Sum along Axis 1: [ 6 15]
Reduce Mean: 3
Reduce Mean along Axis 0: [2 3 4]
Reduce Mean along Axis 1: [2 5]
Reduce Max: 6
Reduce Min: 1
```

Matrix Operations

Matrix operations are fundamental for machine learning algorithms, especially in linear algebra. TensorFlow provides various functions to perform matrix operations efficiently.

```
# Define 2D tensors (matrices)

matrix1 = tf.constant([[1, 2], [3, 4]])

matrix2 = tf.constant([[5, 6], [7, 8]])


# Matrix addition

matrix_add = tf.add(matrix1, matrix2)

print("Matrix Addition:")

print(matrix_add.numpy())


# Matrix multiplication

matrix_mul = tf.matmul(matrix1, matrix2)

print("\nMatrix Multiplication:")

print(matrix_mul.numpy())


# Matrix transpose

matrix_transpose = tf.transpose(matrix1)

print("\nMatrix Transpose:")

print(matrix_transpose.numpy())


# Matrix determinant
```

```
matrix_det = tf.linalg.det(tf.cast(matrix1, tf.float32))

print("\nMatrix Determinant:")

print(matrix_det.numpy())


# Matrix inverse

matrix_inverse = tf.linalg.inv(tf.cast(matrix1, tf.float32))

print("\nMatrix Inverse:")

print(matrix_inverse.numpy())


# Matrix trace

matrix_trace = tf.linalg.trace(matrix1)

print("\nMatrix Trace:")

print(matrix_trace.numpy())
```

```
Matrix Addition:
[[ 6  8]
 [10 12]]

Matrix Multiplication:
[[19 22]
 [43 50]]

Matrix Transpose:
[[1 3]
 [2 4]]

Matrix Determinant:
-2.0

Matrix Inverse:
[[-2.0000002   1.0000001 ]
 [ 1.5000001  -0.50000006]]

Matrix Trace:
5
```

Data Manipulation Operations

TensorFlow provides a wide array of functions for manipulating data, including reshaping, slicing, and concatenating tensors.

Reshaping Tensors

Reshaping is the process of changing the shape of a tensor without changing its data.

Example: Reshaping Tensors

```
import tensorflow as tf


# Define a tensor
tensor = tf.constant([[1, 2, 3], [4, 5, 6]])


# Reshape tensor to 3x2
reshaped_tensor = tf.reshape(tensor, [3, 2])
print("Reshaped Tensor (3x2):")
print(reshaped_tensor.numpy())


# Reshape tensor to 1x6
reshaped_tensor2 = tf.reshape(tensor, [1, 6])
print("\nReshaped Tensor (1x6):")
print(reshaped_tensor2.numpy())
```

```
Reshaped Tensor (3x2):
[[1 2]
 [3 4]
 [5 6]]

Reshaped Tensor (1x6):
[[1 2 3 4 5 6]]
```

Slicing Tensors

Slicing allows you to extract a portion of a tensor.

Example: Slicing Tensors

```
# Define a tensor
tensor = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
# Extract the first two rows

slice1 = tf.slice(tensor, [0, 0], [2, 3])

print("Sliced Tensor (first two rows):")

print(slice1.numpy())


# Extract the first column

slice2 = tf.slice(tensor, [0, 0], [3, 1])

print("\nSliced Tensor (first column):")

print(slice2.numpy())
```

```
Sliced Tensor (first two rows):
[[1 2 3]
 [4 5 6]]

Sliced Tensor (first column):
[[1]
 [4]
 [7]]
```

Concatenating Tensors

Concatenation joins tensors along a specified axis.

Example: Concatenating Tensors

```
# Define two tensors

tensor1 = tf.constant([[1, 2], [3, 4]])

tensor2 = tf.constant([[5, 6], [7, 8]])


# Concatenate along axis 0 (rows)

concat0 = tf.concat([tensor1, tensor2], axis=0)

print("Concatenated Tensor (axis 0):")

print(concat0.numpy())


# Concatenate along axis 1 (columns)

concat1 = tf.concat([tensor1, tensor2], axis=1)
```

print("\nConcatenated Tensor (axis 1):")

print(concat1.numpy())

```
Concatenated Tensor (axis 0):
[[1 2]
 [3 4]
 [5 6]
 [7 8]]

Concatenated Tensor (axis 1):
[[1 2 5 6]
 [3 4 7 8]]
```

Stacking Tensors

Stacking creates a new dimension by stacking tensors along a new axis.

Example: Stacking Tensors

Define two tensors

tensor1 = tf.constant([1, 2])

tensor2 = tf.constant([3, 4])

Stack along a new axis

stacked_tensor = tf.stack([tensor1, tensor2], axis=0)

print("Stacked Tensor (axis 0):")

print(stacked_tensor.numpy())

stacked_tensor2 = tf.stack([tensor1, tensor2], axis=1)

print("\nStacked Tensor (axis 1):")

print(stacked_tensor2.numpy())

```
Stacked Tensor (axis 0):
[[1 2]
 [3 4]]

Stacked Tensor (axis 1):
[[1 3]
 [2 4]]
```

Splitting Tensors

Splitting divides a tensor into multiple sub-tensors.

Example: Splitting Tensors

```
# Define a tensor
tensor = tf.constant([[1, 2, 3], [4, 5, 6]])

# Split into 3 sub-tensors along axis 1
split_tensor = tf.split(tensor, num_or_size_splits=3, axis=1)
print("Split Tensors (axis 1):")
for t in split_tensor:
    print(t.numpy())

# Split into 2 sub-tensors along axis 0
split_tensor2 = tf.split(tensor, num_or_size_splits=2, axis=0)
print("\nSplit Tensors (axis 0):")
for t in split_tensor2:
    print(t.numpy())
```

```
Split Tensors (axis 1):
[[1]
 [4]]
[[2]
 [5]]
[[3]
 [6]]

Split Tensors (axis 0):
[[1 2 3]]
[[4 5 6]]
```

Data Shuffling

Shuffling is used to randomize the order of data elements, which is useful in training machine learning models.

Example: Shuffling Data

Define a tensor

tensor = tf.constant([[1, 2], [3, 4], [5, 6], [7, 8]])

Shuffle the tensor

shuffled_tensor = tf.random.shuffle(tensor)

print("Shuffled Tensor:")

print(shuffled_tensor.numpy())

```
Shuffled Tensor:
[[7 8]
 [5 6]
 [3 4]
 [1 2]]
```

Activation Functions, Convolution Operations, and Recurrent Operations

Activation Functions

Activation functions are crucial in neural networks as they introduce non-linearity, enabling the network to learn complex patterns. TensorFlow provides several commonly used activation functions.

Example: Activation Functions

```
import tensorflow as tf


# Define a tensor
tensor = tf.constant([-1.0, 0.0, 1.0, 2.0])


# ReLU (Rectified Linear Unit)
relu = tf.nn.relu(tensor)
print("ReLU Activation:")
print(relu.numpy())


# Sigmoid
sigmoid = tf.nn.sigmoid(tensor)
print("\nSigmoid Activation:")
print(sigmoid.numpy())


# Tanh (Hyperbolic Tangent)
tanh = tf.nn.tanh(tensor)
print("\nTanh Activation:")
print(tanh.numpy())


# Softmax
softmax = tf.nn.softmax(tensor)
print("\nSoftmax Activation:")
print(softmax.numpy())
```

```
ReLU Activation:
[0. 0. 1. 2.]

Sigmoid Activation:
[0.26894143 0.5        0.7310586  0.8807971 ]

Tanh Activation:
[-0.7615942  0.         0.7615942  0.9640276]

Softmax Activation:
[0.0320586  0.08714432 0.2368828  0.6439142 ]
```

Convolution Operations

Convolution operations are fundamental for processing spatial data, such as images. They apply a filter to an input to create feature maps.

Example: Convolution Operations

```
# Define a 4D tensor for a batch of grayscale images [batch, height, width, channels]
input_tensor = tf.random.normal([1, 5, 5, 1])

# Define a convolutional layer
conv_layer = tf.keras.layers.Conv2D(filters=1, kernel_size=3, strides=1, padding='same')

# Apply convolution
output_tensor = conv_layer(input_tensor)
print("Convolution Output:")
print(output_tensor.numpy())


# Define a max pooling layer
max_pool_layer = tf.keras.layers.MaxPooling2D(pool_size=2, strides=2, padding='same')
```

```
# Apply max pooling
pooled_tensor = max_pool_layer(output_tensor)
print("\nMax Pooling Output:")
print(pooled_tensor.numpy())
```

```
Convolution Output:
[[[[ 0.47250944]
   [ 0.5683058 ]
   [-0.05283204]
   [-0.1426486 ]
   [-0.23279421]]

  [[-0.41876495]
   [ 0.26157668]
   [-0.43551576]
   [-0.6921179 ]
   [-0.2869283 ]]

  [[ 0.40157208]
   [ 0.9677091 ]
   [ 0.3271977 ]
   [ 0.3835646 ]
   [ 0.4445812 ]]

  [[ 1.1500304 ]
   [ 0.9466896 ]
   [-0.10730833]
   [-0.03304466]
   [ 0.02212249]]

  [[-0.07137269]
   [-0.5046127 ]
   [-1.4199206 ]
   [ 0.6419237 ]
   [-0.2353569 ]]]]
```

```
Max Pooling Output:
[[[[ 0.5683058 ]
   [-0.05283204]
   [-0.23279421]]

  [[ 1.1500304 ]
   [ 0.3835646 ]
   [ 0.4445812 ]]

  [[-0.07137269]
   [ 0.6419237 ]
   [-0.2353569 ]]]]
```

[Text Wrapping Break]

Convolution with Padding and Strides

Define another 4D tensor

input_tensor = tf.random.normal([1, 7, 7, 1])

Define a convolutional layer with padding and strides

conv_layer = tf.keras.layers.Conv2D(filters=1, kernel_size=3, strides=2, padding='same')

Apply convolution

output_tensor = conv_layer(input_tensor)

print("Convolution with Padding and Strides Output:")

print(output_tensor.numpy())

```
Convolution with Padding and Strides Output:
[[[[-0.5836452 ]
   [ 1.2665899 ]
   [-1.1384947 ]
   [-0.08779764]]

  [[-0.18475841]
   [ 0.8312385 ]
   [-0.1749598 ]
   [ 0.5754744 ]]

  [[-0.1172349 ]
   [-0.94881946]
   [-0.03914974]
   [-0.54958177]]

  [[-0.7253362 ]
   [ 0.41238517]
   [ 0.4158369 ]
   [-0.71953714]]]]
```

Recurrent Operations

Recurrent operations are used in Recurrent Neural Networks (RNNs), which are suitable for sequence data like time series and text.

Example: Simple RNN

```
# Define a 3D tensor for a batch of sequences [batch, timesteps, features]
input_tensor = tf.random.normal([1, 5, 3])

# Define an RNN layer
rnn_layer = tf.keras.layers.SimpleRNN(units=4)

# Apply RNN
output_tensor = rnn_layer(input_tensor)
print("Simple RNN Output:")
```

print(output_tensor.numpy())

```
Simple RNN Output:
[[ 0.68004084  0.557547    0.5659173  -0.07848052]]
```

Example: LSTM (Long Short-Term Memory)

LSTM is a type of RNN that can capture long-term dependencies.

```
# Define an LSTM layer
lstm_layer = tf.keras.layers.LSTM(units=4)

# Apply LSTM
output_tensor = lstm_layer(input_tensor)
print("LSTM Output:")
print(output_tensor.numpy())
```

```
LSTM Output:
[[-0.20183003 -0.333641    0.27021268 -0.0338501 ]]
```

Example: GRU (Gated Recurrent Unit)

GRU is another type of RNN that is computationally efficient.

```
# Define a GRU layer
gru_layer = tf.keras.layers.GRU(units=4)

# Apply GRU
output_tensor = gru_layer(input_tensor)
print("GRU Output:")
print(output_tensor.numpy())
```

```
GRU Output:
[[ 0.05007443 -0.39168274  0.5931334   0.36166507]]
```

Loss Functions

Loss functions measure the difference between the predicted output and the actual target value. They are crucial for guiding the optimization process during training.

Common Loss Functions

Mean Squared Error (MSE): Used for regression tasks.

Binary Cross-Entropy: Used for binary classification tasks.

Categorical Cross-Entropy: Used for multi-class classification tasks.

Example: Loss Functions

```
import tensorflow as tf

# Define true labels and predicted values
y_true = tf.constant([1.0, 0.0, 1.0, 0.0])
y_pred = tf.constant([0.9, 0.1, 0.8, 0.2])

# Mean Squared Error
mse = tf.keras.losses.MeanSquaredError()
mse_loss = mse(y_true, y_pred)
print("Mean Squared Error Loss:", mse_loss.numpy())

# Binary Cross-Entropy
bce = tf.keras.losses.BinaryCrossentropy()
bce_loss = bce(y_true, y_pred)
print("\nBinary Cross-Entropy Loss:", bce_loss.numpy())

# Categorical Cross-Entropy
y_true_cat = tf.constant([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
y_pred_cat = tf.constant([[0.9, 0.05, 0.05], [0.1, 0.8, 0.1], [0.05, 0.05, 0.9]])
cce = tf.keras.losses.CategoricalCrossentropy()
cce_loss = cce(y_true_cat, y_pred_cat)
print("\nCategorical Cross-Entropy Loss:", cce_loss.numpy())
```

```
Mean Squared Error Loss: 0.025000002

Binary Cross-Entropy Loss: 0.1642519

Categorical Cross-Entropy Loss: 0.14462154
```

Gradient Operations

Gradient operations are used to compute the gradients of the loss function with respect to the model parameters. These gradients are then used to update the parameters during the optimization process.

Example: Gradient Computation

```
# Define a simple linear model

class SimpleLinearModel(tf.keras.Model):

    def __init__(self):

        super(SimpleLinearModel, self).__init__()

        self.dense = tf.keras.layers.Dense(units=1, input_shape=(1,))


    def call(self, inputs):

        return self.dense(inputs)


# Create a model instance

model = SimpleLinearModel()


# Define inputs and targets

inputs = tf.constant([[1.0], [2.0], [3.0], [4.0]])

targets = tf.constant([[2.0], [3.0], [4.0], [5.0]])


# Define a loss function

loss_fn = tf.keras.losses.MeanSquaredError()


# Use GradientTape to record the operations
```

```
with tf.GradientTape() as tape:
    predictions = model(inputs)
    loss = loss_fn(targets, predictions)

# Compute gradients
gradients = tape.gradient(loss, model.trainable_variables)
print("\nGradients:")
for var, grad in zip(model.trainable_variables, gradients):
    print(f"{var.name}: {grad.numpy()}")
```

```
Gradients:
simple_linear_model/dense/kernel:0: [[-17.51575]]
simple_linear_model/dense/bias:0: [-6.1719174]
```

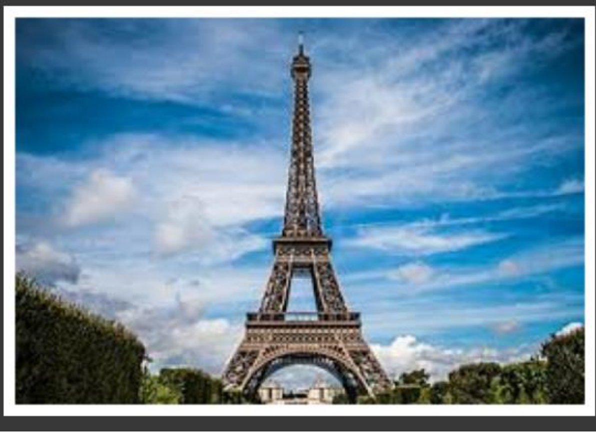
Image Operations

TensorFlow provides various functions for image manipulation, including resizing, cropping, flipping, and more.

Loading and Displaying an Image

```
import tensorflow as tf
import matplotlib.pyplot as plt

# Load an image from a file
image_path =' /content/download (1).jpg'

image = tf.io.read_file(image_path)
image = tf.image.decode_jpeg(image, channels=3)

# Display the image
plt.imshow(image.numpy())
plt.axis('off')
plt.show()
```

```
[ ] import tensorflow as tf
    import matplotlib.pyplot as plt

    # Load an image from a file
    image_path = '/content/download.jpg'
    image = tf.io.read_file(image_path)
    image = tf.image.decode_jpeg(image, channels=3)

    # Display the image
    plt.imshow(image.numpy())
    plt.axis('off')
    plt.show()
```

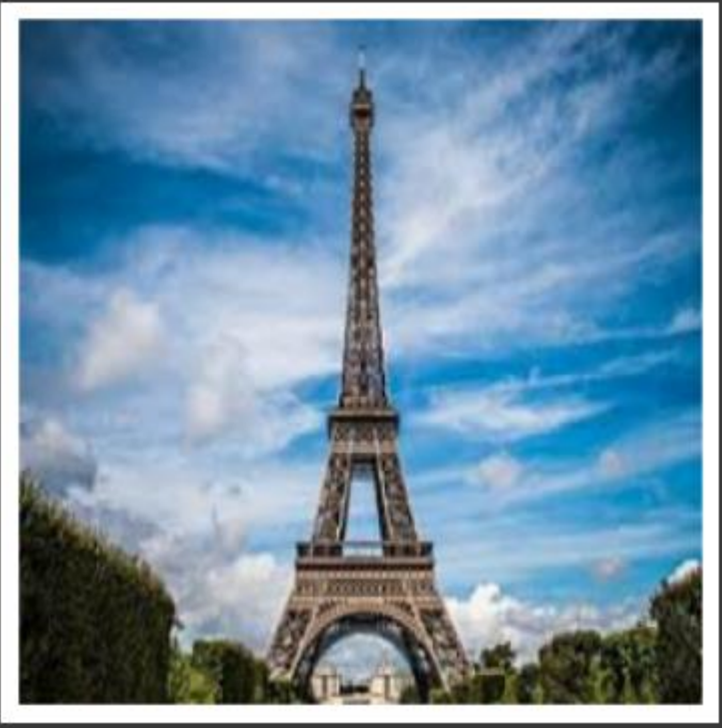


Resizing an Image

Resizing is a common preprocessing step to ensure all images are of the same size.

```
# Resize the image to 256x256

resized_image = tf.image.resize(image, [256, 256])

# Display the resized image

plt.imshow(resized_image.numpy().astype("uint8"))

plt.axis('off')

plt.show()
```

```
# Resize the image to 256x256
resized_image = tf.image.resize(image, [256, 256])

# Display the resized image
plt.imshow(resized_image.numpy().astype("uint8"))
plt.axis('off')
plt.show()
```

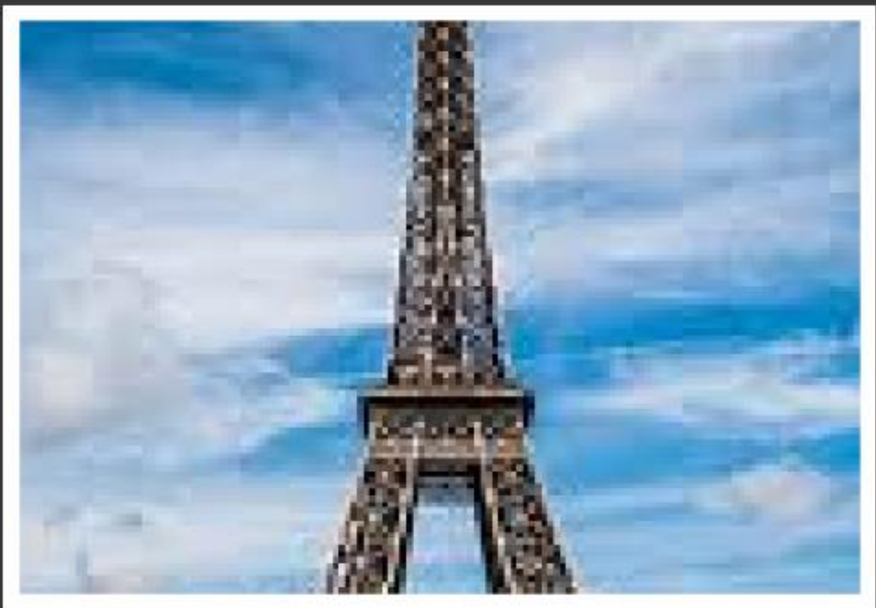


Cropping an Image

Cropping is used to extract a specific region of the image.

```
# Crop the central part of the image

cropped_image = tf.image.central_crop(image, central_fraction=0.5)

# Display the cropped image

plt.imshow(cropped_image.numpy().astype("uint8"))

plt.axis('off')

plt.show()
```

```python
# Crop the central part of the image
cropped_image = tf.image.central_crop(image, central_fraction=0.5)

# Display the cropped image
plt.imshow(cropped_image.numpy().astype("uint8"))
plt.axis('off')
plt.show()
```

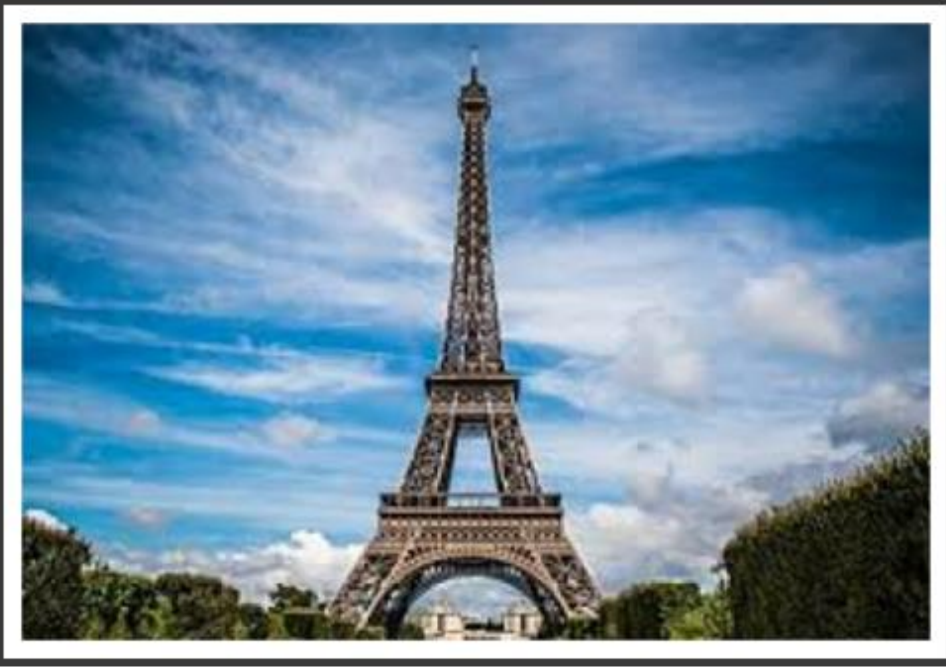


Flipping an Image

Flipping can be used for data augmentation to improve model generalization.

```python
# Flip the image horizontally

flipped_image = tf.image.flip_left_right(image)

# Display the flipped image

plt.imshow(flipped_image.numpy().astype("uint8"))

plt.axis('off')

plt.show()
```

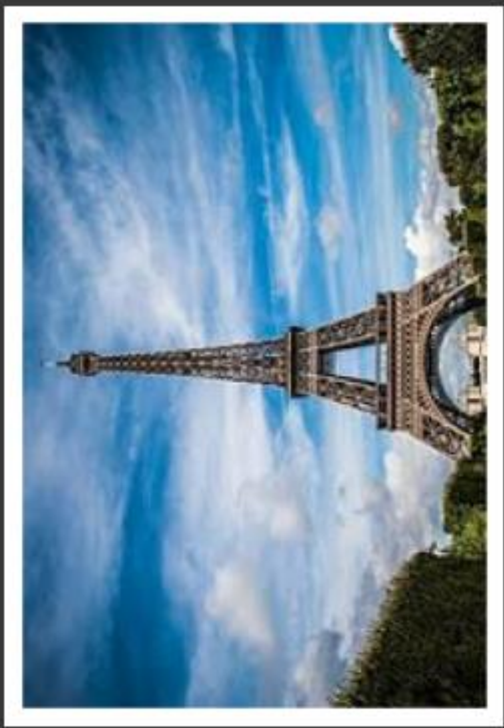
```python
# Flip the image horizontally
flipped_image = tf.image.flip_left_right(image)

# Display the flipped image
plt.imshow(flipped_image.numpy().astype("uint8"))
plt.axis('off')
plt.show()
```



Rotating an Image

Rotating images can also be used for data augmentation.

```
[ ] # Rotate the image by 90 degrees
    rotated_image = tf.image.rot90(image)

    # Display the rotated image
    plt.imshow(rotated_image.numpy().astype("uint8"))
    plt.axis('off')
    plt.show()
```



There many other   u can go through it