```
In [ ]: from google.colab import files
        files.upload()  # This will prompt you to upload `kaggle.json`
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving kaggle.json to kaggle.json
```


```
Out[ ]: {'kaggle.json': b'{"username":"satchibaghla","key":"3e1c74debb59372a257006d71ddba3f4"}'}
```


```
In [5]: # Make a Kaggle directory and move the kaggle.json file
        !mkdir -p ~/.kaggle
        !cp kaggle.json ~/.kaggle/
        !chmod 600 ~/.kaggle/kaggle.json

        # Download the Intel Image Classification dataset from Kaggle
        !kaggle datasets download -d puneet6060/intel-image-classification

        # Unzip the dataset
        !unzip intel-image-classification.zip
```

```
Dataset URL: https://www.kaggle.com/datasets/puneet6060/intel-image-classification
License(s): copyright-authors
intel-image-classification.zip: Skipping, found more recently modified local copy (use --force to force download)
Archive:  intel-image-classification.zip
replace seg_pred/seg_pred/10004.jpg? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

STEP 1: Data Loading and Visualization

Use Keras' ImageDataGenerator to load images and apply augmentation for better generalization.

```
In [6]: import tensorflow as tf
        from tensorflow.keras.preprocessing.image import ImageDataGenerator

        # Define paths to train and test directories
        train_dir = 'seg_train/seg_train'
        test_dir = 'seg_test/seg_test'

        # Set up ImageDataGenerator for training and testing
        train_datagen = ImageDataGenerator(rescale=1./255, validation_split=0.2,
                                           rotation_range=40, width_shift_range=0.2,
                                           shear_range=0.2, zoom_range=0.2, horizont

        test_datagen = ImageDataGenerator(rescale=1./255)

        # Load training and validation data
        train_data = train_datagen.flow_from_directory(train_dir, target_size=(150,
        validation_data = train_datagen.flow_from_directory(train_dir, target_size=(

        # Load test data
        test_data = test_datagen.flow_from_directory(test_dir, target_size=(150, 150
```

```
Found 11230 images belonging to 6 classes.
Found 2804 images belonging to 6 classes.
Found 3000 images belonging to 6 classes.
```

Explanation:

ImageDataGenerator: This is used to load images from the directory and apply preprocessing (in this case, just rescaling).
flow_from_directory: Loads images from the directory and assigns labels based on folder names.
We visualize 9 random images from the training set using matplotlib.

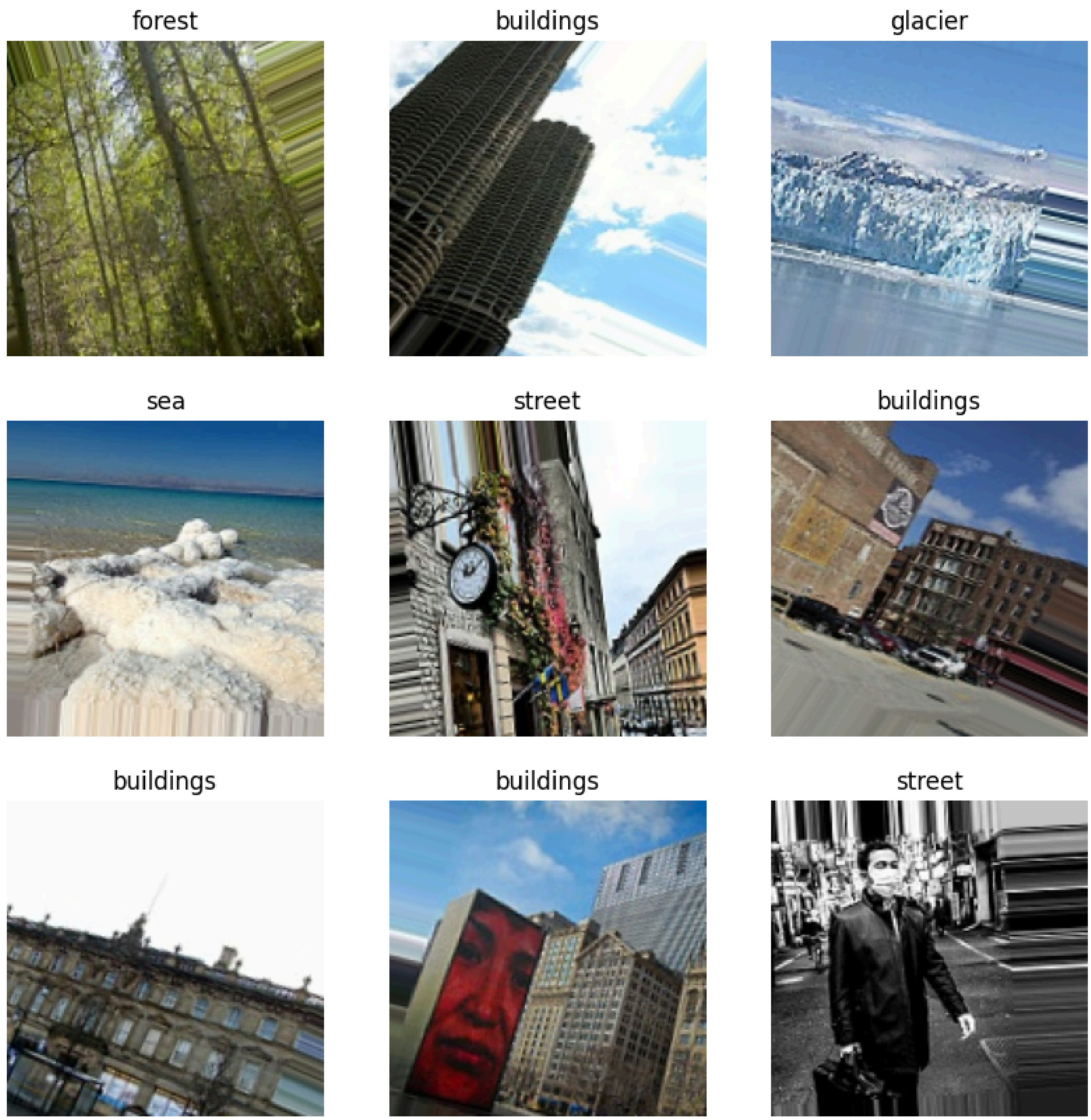
In [7]:

```python
import matplotlib.pyplot as plt
import numpy as np

class_names = list(train_data.class_indices.keys())

# Retrieve a batch of images and labels from the training data
images, labels = next(train_data)

# Plot some of the images
plt.figure(figsize=(10, 10))
for i in range(9):
    plt.subplot(3, 3, i + 1)
    plt.imshow(images[i])
    plt.title(class_names[np.argmax(labels[i])])
    plt.axis('off')
plt.show()
```



STEP 2: Model Architecture

Define a CNN architecture with convolutional layers, max pooling, batch normalization, and dropout.

In [8]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dr

# Define the CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D((2, 2)),
    BatchNormalization(),

    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    BatchNormalization(),
```

```
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    BatchNormalization(),

    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(6, activation='softmax')  # 6 classes in the dataset
])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['a

# Display the model summary
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 148, 148, 32)      896

 max_pooling2d (MaxPooling2  (None, 74, 74, 32)        0
 D)

 batch_normalization (Batch  (None, 74, 74, 32)        128
 Normalization)

 conv2d_1 (Conv2D)           (None, 72, 72, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 36, 36, 64)        0
 g2D)

 batch_normalization_1 (Bat  (None, 36, 36, 64)        256
 chNormalization)

 conv2d_2 (Conv2D)           (None, 34, 34, 128)       73856

 max_pooling2d_2 (MaxPoolin  (None, 17, 17, 128)       0
 g2D)

 batch_normalization_2 (Bat  (None, 17, 17, 128)       512
 chNormalization)

 flatten (Flatten)           (None, 36992)             0

 dense (Dense)               (None, 256)               9470208

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 6)                 1542

=================================================================
Total params: 9565894 (36.49 MB)
Trainable params: 9565446 (36.49 MB)
Non-trainable params: 448 (1.75 KB)
_________________________________________________________________
```

Explanation:

We use Conv2D layers with ReLU activation, followed by MaxPooling2D to reduce spatial dimensions.
BatchNormalization helps in stabilizing and speeding up training.
The fully connected (dense) layers at the end allow the model to make predictions based on the learned features.
We use softmax activation in the final layer since this is a multi-class classification problem.

STEP 3: Model Training

Train the model on the training data and validate it on the validation data.

In [9]:
```
# Train the model with training and validation data
history = model.fit(
    train_data,
    validation_data=validation_data,
    steps_per_epoch=train_data.samples // 32,
    validation_steps=validation_data.samples // 32,
    epochs=20
)
```

```
Epoch 1/20
350/350 [==============================] - 96s 270ms/step - loss: 5.4759 - a
ccuracy: 0.4294 - val_loss: 2.4049 - val_accuracy: 0.3825
Epoch 2/20
350/350 [==============================] - 94s 268ms/step - loss: 1.5079 - a
ccuracy: 0.5143 - val_loss: 1.2872 - val_accuracy: 0.5269
Epoch 3/20
350/350 [==============================] - 94s 268ms/step - loss: 1.1509 - a
ccuracy: 0.5699 - val_loss: 0.9456 - val_accuracy: 0.6498
Epoch 4/20
350/350 [==============================] - 93s 266ms/step - loss: 1.0941 - a
ccuracy: 0.5932 - val_loss: 0.9199 - val_accuracy: 0.6494
Epoch 5/20
350/350 [==============================] - 93s 267ms/step - loss: 1.0641 - a
ccuracy: 0.6050 - val_loss: 0.9405 - val_accuracy: 0.6724
Epoch 6/20
350/350 [==============================] - 93s 266ms/step - loss: 1.0376 - a
ccuracy: 0.6204 - val_loss: 1.0035 - val_accuracy: 0.6455
Epoch 7/20
350/350 [==============================] - 93s 265ms/step - loss: 1.0051 - a
ccuracy: 0.6234 - val_loss: 1.0657 - val_accuracy: 0.6200
Epoch 8/20
350/350 [==============================] - 94s 269ms/step - loss: 0.9785 - a
ccuracy: 0.6407 - val_loss: 0.8951 - val_accuracy: 0.6642
Epoch 9/20
350/350 [==============================] - 94s 268ms/step - loss: 0.9578 - a
ccuracy: 0.6498 - val_loss: 0.8715 - val_accuracy: 0.6828
Epoch 10/20
350/350 [==============================] - 93s 266ms/step - loss: 0.9325 - a
ccuracy: 0.6557 - val_loss: 1.1449 - val_accuracy: 0.5858
Epoch 11/20
350/350 [==============================] - 94s 268ms/step - loss: 0.9285 - a
ccuracy: 0.6631 - val_loss: 1.0149 - val_accuracy: 0.6422
Epoch 12/20
350/350 [==============================] - 94s 269ms/step - loss: 0.9125 - a
ccuracy: 0.6738 - val_loss: 0.8896 - val_accuracy: 0.6800
Epoch 13/20
350/350 [==============================] - 93s 265ms/step - loss: 0.8736 - a
ccuracy: 0.6836 - val_loss: 0.8427 - val_accuracy: 0.6997
Epoch 14/20
350/350 [==============================] - 94s 269ms/step - loss: 0.8664 - a
ccuracy: 0.6831 - val_loss: 0.8889 - val_accuracy: 0.6731
Epoch 15/20
350/350 [==============================] - 94s 267ms/step - loss: 0.8296 - a
ccuracy: 0.7018 - val_loss: 1.1103 - val_accuracy: 0.6494
Epoch 16/20
350/350 [==============================] - 93s 265ms/step - loss: 0.8144 - a
ccuracy: 0.7161 - val_loss: 0.8949 - val_accuracy: 0.6778
Epoch 17/20
350/350 [==============================] - 94s 269ms/step - loss: 0.8035 - a
ccuracy: 0.7178 - val_loss: 0.9964 - val_accuracy: 0.6239
Epoch 18/20
350/350 [==============================] - 94s 268ms/step - loss: 0.7787 - a
ccuracy: 0.7264 - val_loss: 1.1395 - val_accuracy: 0.6096
Epoch 19/20
350/350 [==============================] - 94s 268ms/step - loss: 0.7712 - a
```

```
ccuracy: 0.7324 - val_loss: 0.7498 - val_accuracy: 0.7446
Epoch 20/20
350/350 [==============================] - 93s 267ms/step - loss: 0.7387 - a
ccuracy: 0.7402 - val_loss: 0.9594 - val_accuracy: 0.6789
```

model.fit() trains the CNN using the training.

STEP 4: Model Evaluation

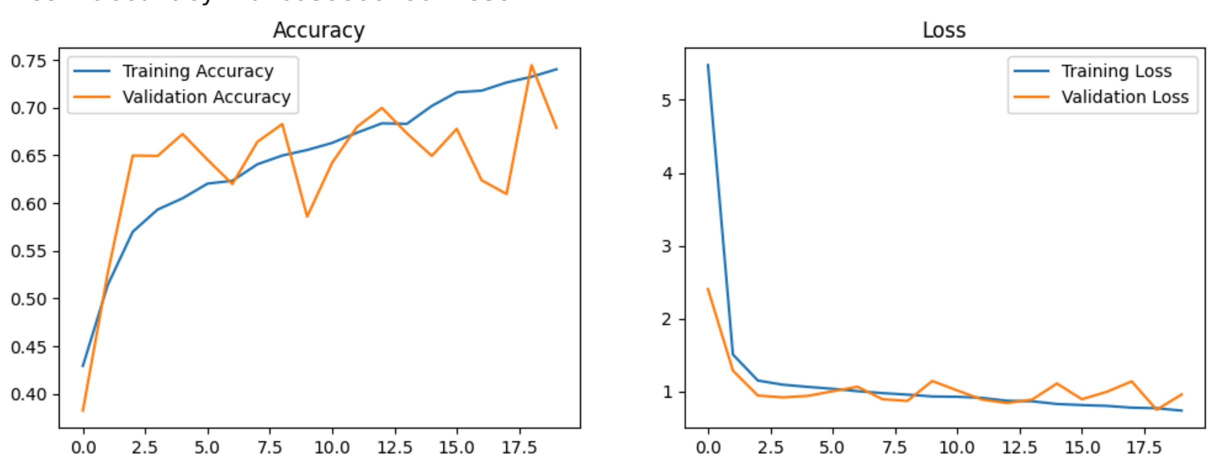
Evaluate the model on the test set and plot training and validation accuracy/loss.

In [10]:
```python
# Evaluate on test data
test_loss, test_acc = model.evaluate(test_data)
print(f"Test accuracy: {test_acc}")

# Plot accuracy and loss curves
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.title('Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.title('Loss')
plt.show()
```

```
94/94 [==============================] - 4s 43ms/step - loss: 1.0381 - accur
acy: 0.6960
Test accuracy: 0.6959999799728394
```



We evaluate the trained model on the test data to get the test accuracy.
The accuracy and loss curves help visualize the model's performance across epochs.

STEP 5: Display Confusion Matrix

Analyze the model’s predictions on the test set with a confusion matrix.

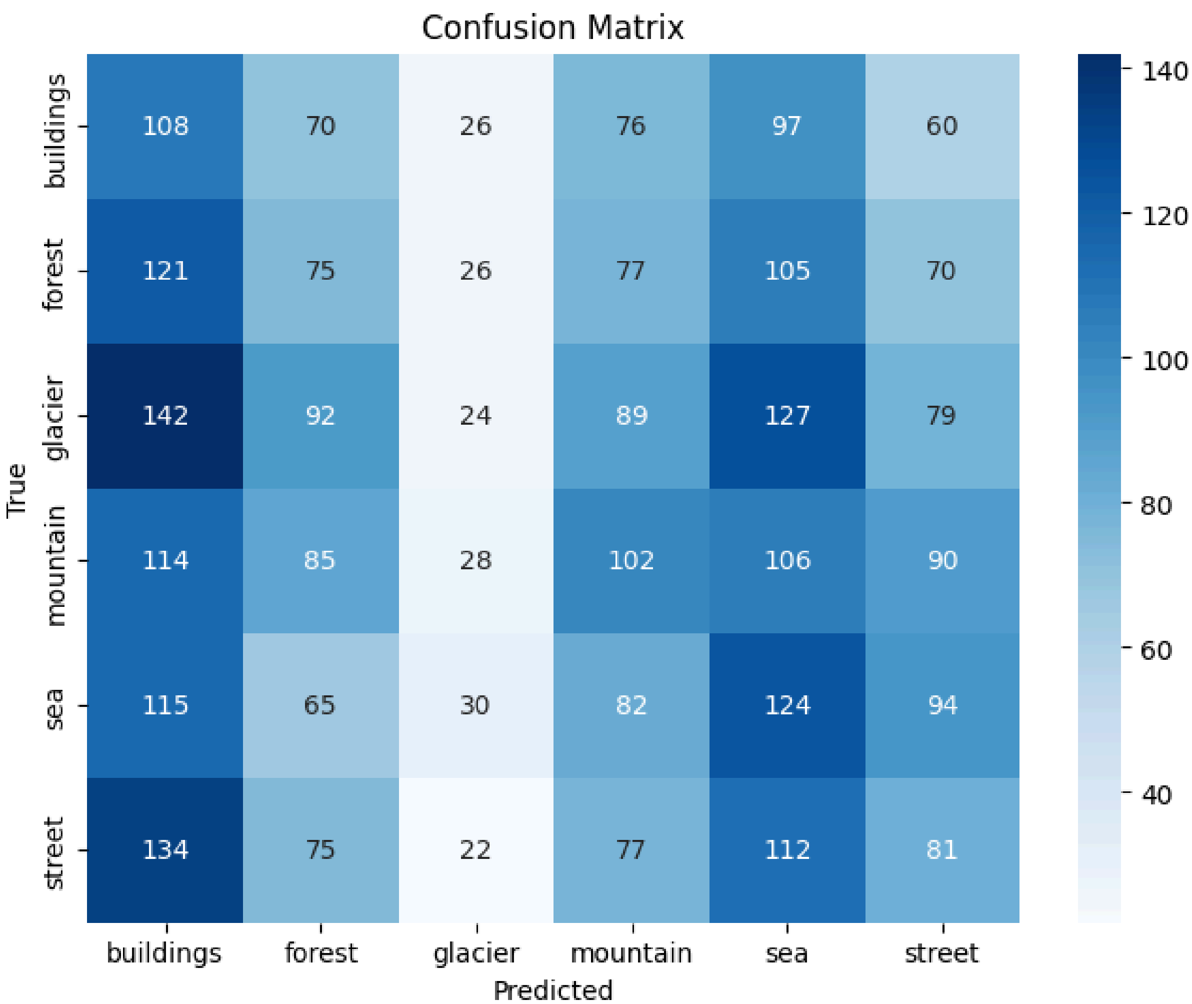
In [11]:

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Get predictions on test data
y_pred = np.argmax(model.predict(test_data), axis=-1)
y_true = test_data.classes

# Plot confusion matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_names,
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix")
plt.show()
```

```
94/94 [==============================] - 4s 44ms/step
```



STEP 5: Optimization

1. Experiment with Data Augmentation
2. Fine-Tune Hyperparameters
   2.1. Adjust Learning Rate and Batch Size
   2.2. Modify the Number of Filters

In [ ]:
```
# Define ImageDataGenerator with additional augmentation techniques
train_datagen = ImageDataGenerator(
    rescale=1./255,            # Rescale pixel values
    validation_split=0.2,       # Split for validation set
    rotation_range=40,         # Random rotations
    width_shift_range=0.2,     # Randomly shift images horizontally
    height_shift_range=0.2,    # Randomly shift images vertically
    shear_range=0.2,           # Shear transformations
    zoom_range=0.2,            # Random zoom
    horizontal_flip=True,      # Randomly flip images
    fill_mode='nearest'        # Fill mode for new pixels
)

# Load training and validation data with the updated augmentation
train_data = train_datagen.flow_from_directory(
    train_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='categorical',
    subset='training'
)

validation_data = train_datagen.flow_from_directory(
    train_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='categorical',
    subset='validation'
)
```

In [ ]:
```
from tensorflow.keras.optimizers import Adam

# Compile the model with a different learning rate
learning_rate = 1e-4  # Reduced learning rate
model.compile(optimizer=Adam(learning_rate=learning_rate),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Re-train the model with a different batch size
history = model.fit(
    train_data,
    validation_data=validation_data,
    steps_per_epoch=train_data.samples // 32,  # Adjust this according to th
    validation_steps=validation_data.samples // 32,
    epochs=20
)
```

In [ ]:

```
# Define a new CNN model with different filter sizes
model = Sequential([
    Conv2D(64, (3, 3), activation='relu', input_shape=(150, 150, 3)),  # Inc
    MaxPooling2D((2, 2)),
    BatchNormalization(),

    Conv2D(128, (3, 3), activation='relu'),  # Increased filters
    MaxPooling2D((2, 2)),
    BatchNormalization(),

    Conv2D(256, (3, 3), activation='relu'),  # Increased filters
    MaxPooling2D((2, 2)),
    BatchNormalization(),

    Flatten(),
    Dense(512, activation='relu'),  # Increased dense layer size
    Dropout(0.5),
    Dense(6, activation='softmax')  # Output layer for 6 classes
])

# Compile the new model
model.compile(optimizer=Adam(learning_rate=learning_rate),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Re-train the model
history = model.fit(
    train_data,
    validation_data=validation_data,
    steps_per_epoch=train_data.samples // 32,
    validation_steps=validation_data.samples // 32,
    epochs=20
)
```