arunothsymena-215-lab5

October 22, 2024

1 Neural Networks and Deep Learning

Arunoth Symen A
2347215

Lab Assignment: Implementing CNN on the Fashion-MNIST Dataset.

Objective: In this lab, you will implement a Convolutional Neural Network (CNN) using the Intel Image Classification dataset. Your task is to train a CNN model to classify these images with high accuracy.

Step 1: Import Necessary Libraries

[20]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,␣
↪Dropout, BatchNormalization
from sklearn.metrics import confusion_matrix
```

This step involves importing necessary libraries such as TensorFlow, Matplotlib, and Seaborn, which provide functions for building neural networks, visualizing data, and performing statistical analyses.

Step 2: Load and Visualize the Dataset

[22]:
```
import kagglehub

# Download the dataset
path = kagglehub.dataset_download("puneet6060/intel-image-classification")
print("Path to dataset files:", path)
```

```
Downloading from
https://www.kaggle.com/api/v1/datasets/download/puneet6060/intel-image-
classification?dataset_version_number=2…
```

```
100%|     | 346M/346M [00:23<00:00, 15.3MB/s]
```

```
Extracting files…


Path to dataset files:
C:\Users\Kalpana\.cache\kagglehub\datasets\puneet6060\intel-image-
classification\versions\2
```

Here, the Intel Image Classification dataset is downloaded and unzipped using Kaggle's API, making the images available for further processing and analysis.

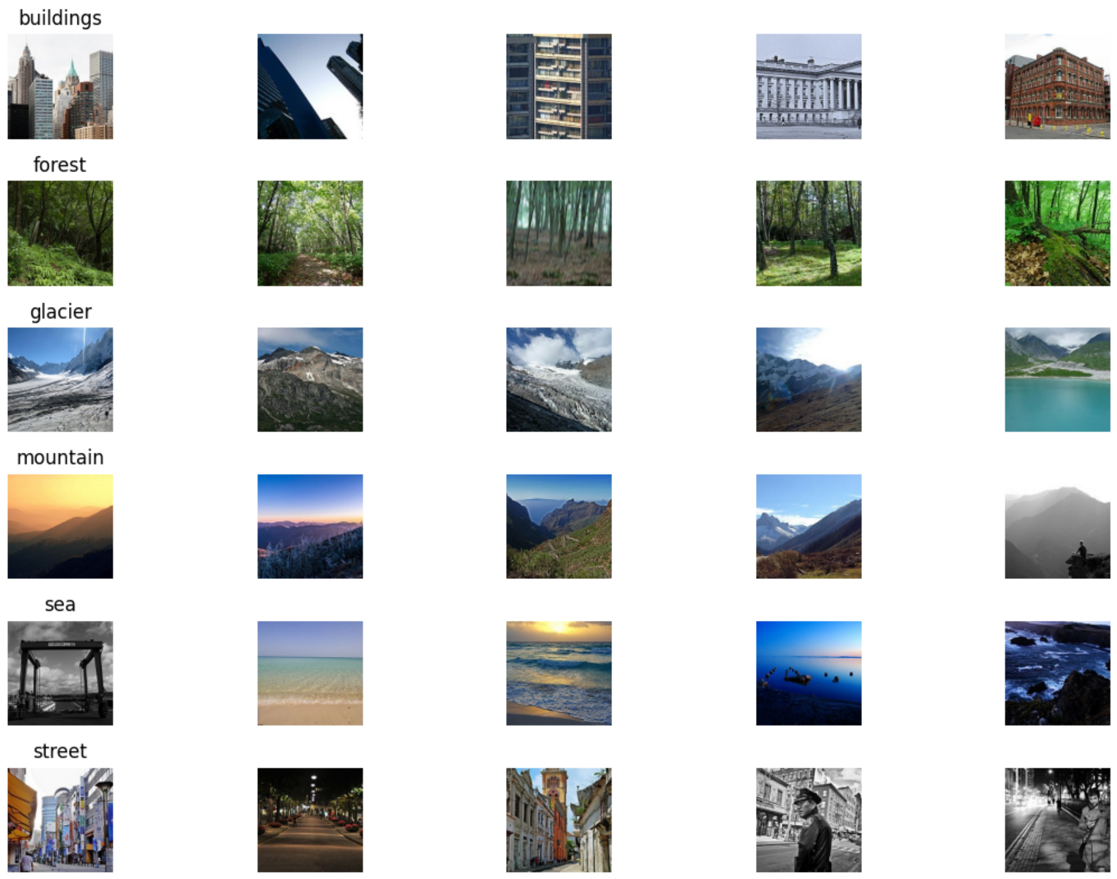
Step 3: Dataset Overview and EDA

1. Visualize a Few Samples

[23]:
```
# Define the data directory
data_dir = 'c:/Users/Kalpana/Documents/NN&DL/intel-image-classification/
 ↪seg_train/seg_train'

# Class names
class_names = os.listdir(data_dir)

# Plot a few samples from each class
plt.figure(figsize=(12, 8))
for i, class_name in enumerate(class_names):
    img_path = os.path.join(data_dir, class_name)
    images = os.listdir(img_path)[:5]  # First 5 images
    for j, img in enumerate(images):
        img_full_path = os.path.join(img_path, img)
        img_array = plt.imread(img_full_path)
        plt.subplot(len(class_names), 5, i * 5 + j + 1)
        plt.imshow(img_array)
        plt.axis('off')
        if j == 0:
            plt.title(class_name)
plt.tight_layout()
plt.show()
```

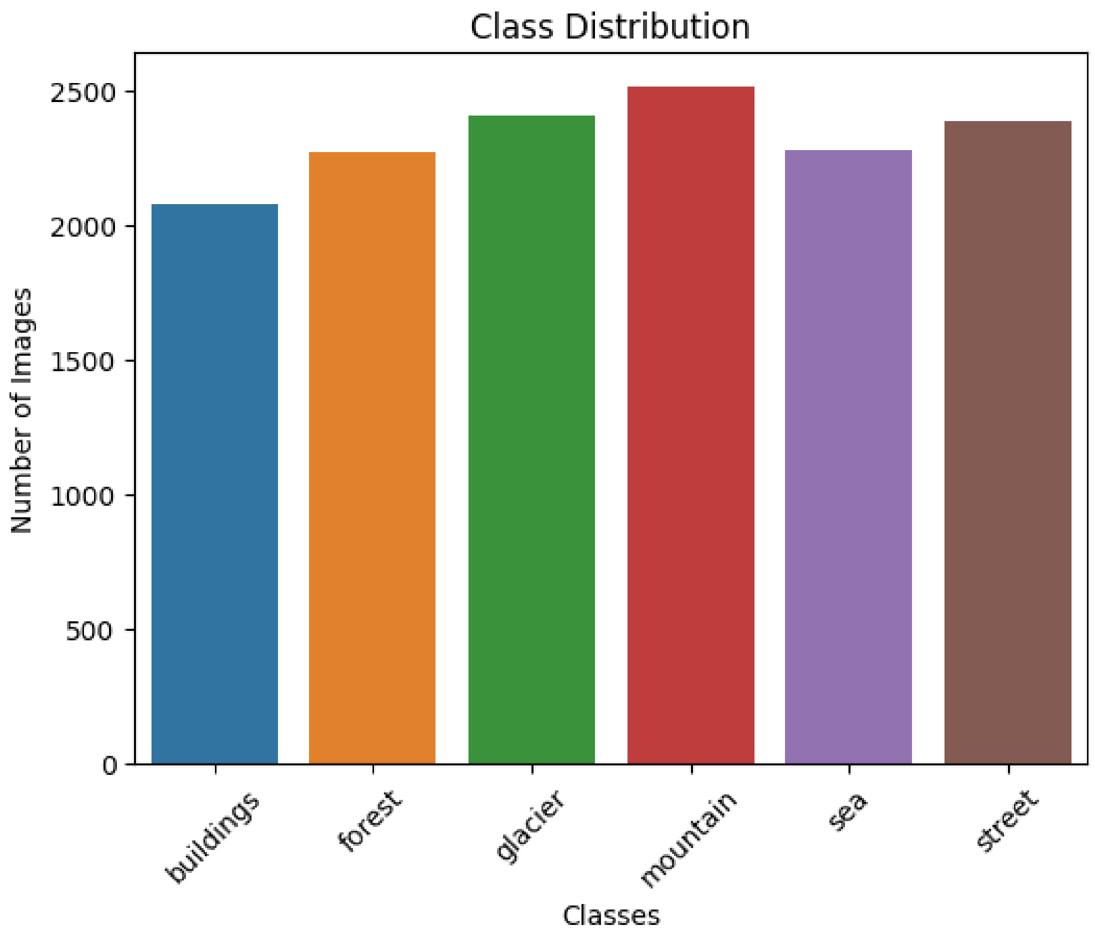


In this step, we visualize a few samples from each class and plot the class distribution to gain insights into the dataset's structure, ensuring a balanced representation across categories.

2. Dataset Statistics

[25]:
```
# Count the number of images in each class
class_counts = {}
for class_name in class_names:
    class_path = os.path.join(data_dir, class_name)
    class_counts[class_name] = len(os.listdir(class_path))

# Display the class distribution
sns.barplot(x=list(class_counts.keys()), y=list(class_counts.values()))
plt.title('Class Distribution')
plt.xlabel('Classes')
plt.ylabel('Number of Images')
plt.xticks(rotation=45)
plt.show()
```



Step 4: Preprocess the Data

1. Create Image Data Generators

[26]:
```
# Create ImageDataGenerators
train_datagen = ImageDataGenerator(
    rescale=1./255,
    validation_split=0.2  # Split the data for validation
)

# Training generator
train_generator = train_datagen.flow_from_directory(
    data_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='categorical',
    subset='training'
)
```

```
# Validation generator
validation_generator = train_datagen.flow_from_directory(
    data_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='categorical',
    subset='validation'
)
```

```
Found 11139 images belonging to 6 classes.
Found 2781 images belonging to 6 classes.
```

We create ImageDataGenerator objects to normalize pixel values and split the data into training and validation sets, while also applying basic data augmentation techniques to improve model generalization.

Step 5: Model Architecture

[27]:
```
# Define the CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D(),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(len(class_names), activation='softmax')  # Output layer for the␣
 ↪number of classes
])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])
```

```
C:\Users\Kalpana\AppData\Roaming\Python\Python311\site-
packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

A Convolutional Neural Network (CNN) is defined with multiple convolutional and pooling layers, followed by dense layers, to learn hierarchical features from the input images and classify them into distinct categories.

Step 6: Train the Model

[28]:
```
# Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // train_generator.batch_size,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // validation_generator.
 ↪batch_size,
    epochs=10  # Adjust based on your needs
)
```

```
Epoch 1/10

C:\Users\Kalpana\AppData\Roaming\Python\Python311\site-
packages\keras\src\trainers\data_adapters\py_dataset_adapter.py:121:
UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in
its constructor. `**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will be
ignored.
  self._warn_if_super_not_called()

348/348               120s 341ms/step -
accuracy: 0.4411 - loss: 1.3873 - val_accuracy: 0.6766 - val_loss: 0.8686
Epoch 2/10
348/348               0s 269us/step -
accuracy: 0.6562 - loss: 0.8653 - val_accuracy: 0.5862 - val_loss: 0.9917
Epoch 3/10

c:\Program Files\Python311\Lib\contextlib.py:155: UserWarning: Your input ran
out of data; interrupting training. Make sure that your dataset or generator can
generate at least `steps_per_epoch * epochs` batches. You may need to use the
`.repeat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)

348/348               114s 328ms/step -
accuracy: 0.6688 - loss: 0.8732 - val_accuracy: 0.7297 - val_loss: 0.7320
Epoch 4/10
348/348               0s 255us/step -
accuracy: 0.6875 - loss: 0.9776 - val_accuracy: 0.8621 - val_loss: 0.4922
Epoch 5/10
348/348               116s 332ms/step -
accuracy: 0.7538 - loss: 0.7064 - val_accuracy: 0.7903 - val_loss: 0.5784
Epoch 6/10
348/348               0s 249us/step -
accuracy: 0.8438 - loss: 0.6073 - val_accuracy: 0.7586 - val_loss: 0.7076
Epoch 7/10
348/348               118s 339ms/step -
accuracy: 0.7915 - loss: 0.5808 - val_accuracy: 0.8212 - val_loss: 0.5188
Epoch 8/10
348/348               0s 264us/step -
```

```
accuracy: 0.8750 - loss: 0.3987 - val_accuracy: 0.8276 - val_loss: 1.0367
Epoch 9/10
348/348              110s 314ms/step -
accuracy: 0.8234 - loss: 0.4952 - val_accuracy: 0.8219 - val_loss: 0.5319
Epoch 10/10
348/348              0s 252us/step -
accuracy: 0.7500 - loss: 0.6177 - val_accuracy: 0.7241 - val_loss: 0.5439
```

The model is trained on the training data using the augmented data generator, allowing it to learn from a diverse set of images, while validation metrics are monitored to prevent overfitting.

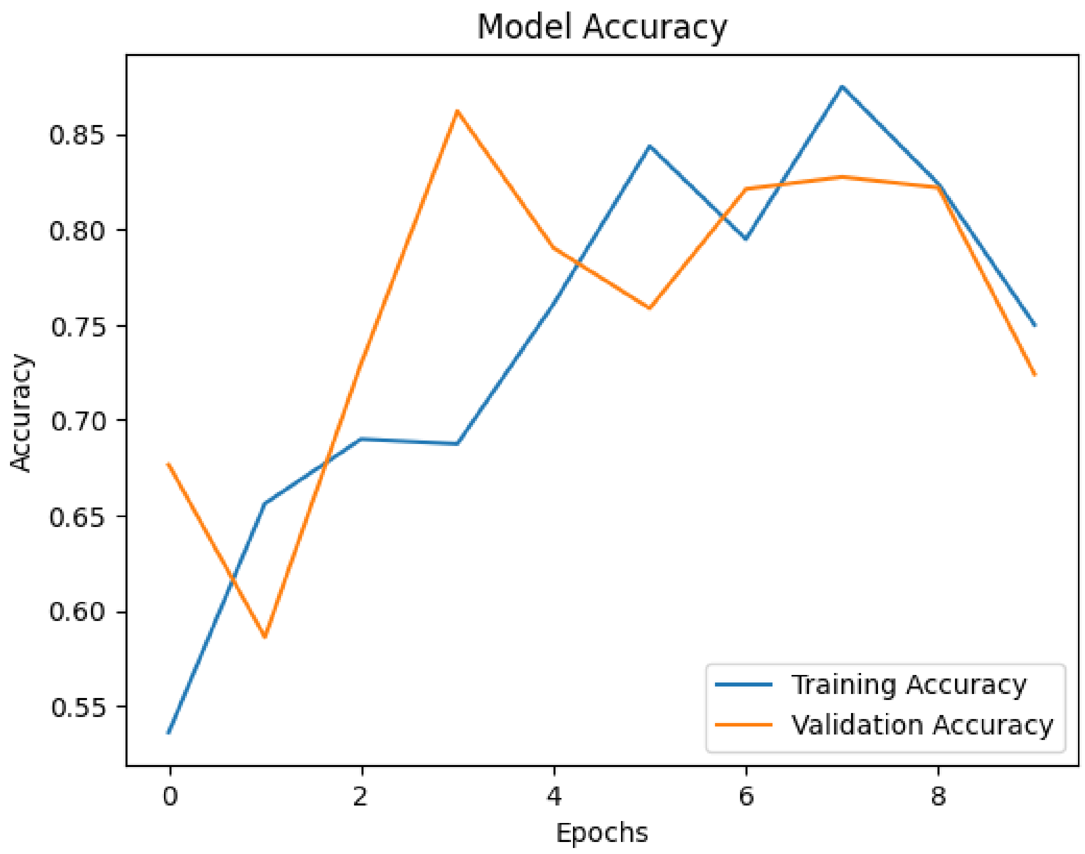
Step 7: Evaluate the Model

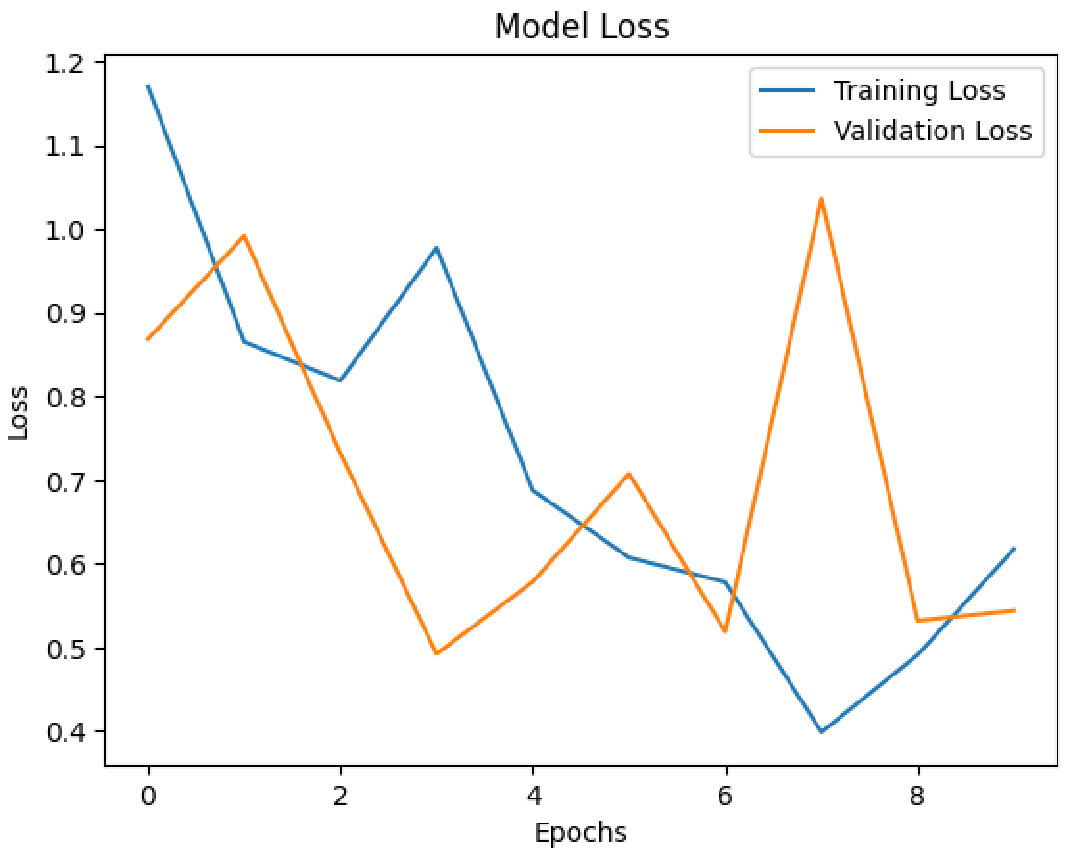
[29]:
```python
# Evaluate the model
test_loss, test_acc = model.evaluate(validation_generator)
print("Validation Accuracy: ", test_acc)

# Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Plot training and validation loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
87/87              7s 79ms/step -
accuracy: 0.8209 - loss: 0.5363
Validation Accuracy:  0.816253125667572
```


Model Accuracy
Accuracy
0.55
0.60
0.65
0.70
0.75
0.80
0.85
Epochs
0
2
4
6
8
Training Accuracy
Validation Accuracy



After training, the model's performance is evaluated on the validation set, and accuracy and loss curves are plotted to visualize how well the model has learned and generalized to unseen data.

Step 8: Analyze Misclassifications with a Confusion Matrix

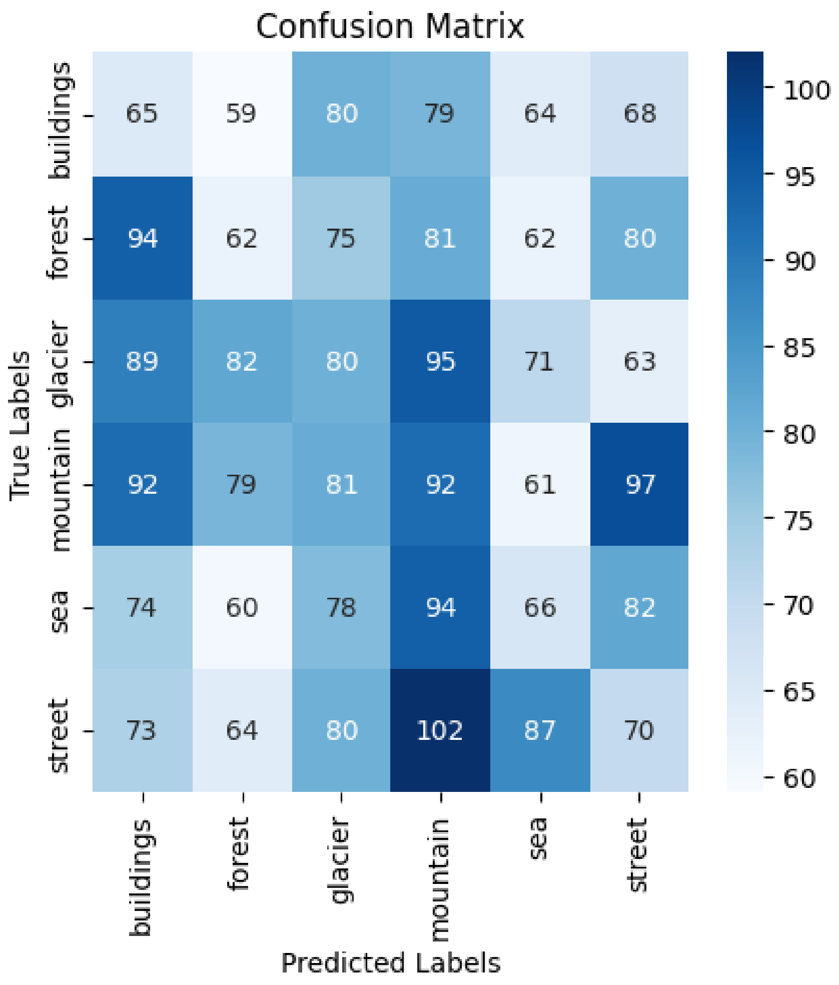
[31]:
```
# Get predictions
y_pred = model.predict(validation_generator)
y_pred_classes = np.argmax(y_pred, axis=1)

# Get true labels
y_true = validation_generator.classes

# Confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred_classes)

# Plot confusion matrix
plt.figure(figsize=(5, 5))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',␣
 ↪xticklabels=class_names, yticklabels=class_names)
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

```
87/87                7s 79ms/step
```



A confusion matrix is generated to analyze misclassified samples, providing insights into which classes the model struggles to differentiate, which can inform future improvements.

Step 9: Data Augmentation

[32]:
```
# Augmented ImageDataGenerator
train_datagen_augmented = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest',
    validation_split=0.2
)

# Create augmented training and validation generators
train_generator_augmented = train_datagen_augmented.flow_from_directory(
    data_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='categorical',
    subset='training'
)

validation_generator_augmented = train_datagen_augmented.flow_from_directory(
    data_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='categorical',
    subset='validation'
)

# Train the model with augmented data
history_augmented = model.fit(
    train_generator_augmented,
    steps_per_epoch=train_generator_augmented.samples //␣
 ↪train_generator_augmented.batch_size,
    validation_data=validation_generator_augmented,
    validation_steps=validation_generator_augmented.samples //␣
 ↪validation_generator_augmented.batch_size,
    epochs=10  # Adjust as needed
)
```

```
Found 11139 images belonging to 6 classes.
Found 2781 images belonging to 6 classes.
Epoch 1/10
```

```
C:\Users\Kalpana\AppData\Roaming\Python\Python311\site-
packages\keras\src\trainers\data_adapters\py_dataset_adapter.py:121:
UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in
its constructor. `**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will be
ignored.
  self._warn_if_super_not_called()
```

```
348/348             140s 400ms/step -
```

```
accuracy: 0.7030 - loss: 0.8296 - val_accuracy: 0.7482 - val_loss: 0.6749
Epoch 2/10
348/348              0s 245us/step -
accuracy: 0.8750 - loss: 0.4376 - val_accuracy: 0.7931 - val_loss: 0.5468
Epoch 3/10
```

```
c:\Program Files\Python311\Lib\contextlib.py:155: UserWarning: Your input ran
out of data; interrupting training. Make sure that your dataset or generator can
generate at least `steps_per_epoch * epochs` batches. You may need to use the
`.repeat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
```

```
348/348              137s 390ms/step -
accuracy: 0.7346 - loss: 0.7483 - val_accuracy: 0.7903 - val_loss: 0.5986
Epoch 4/10
348/348              0s 245us/step -
accuracy: 0.8438 - loss: 0.5073 - val_accuracy: 0.7931 - val_loss: 0.4354
Epoch 5/10
348/348              138s 393ms/step -
accuracy: 0.7497 - loss: 0.6965 - val_accuracy: 0.7867 - val_loss: 0.5900
Epoch 6/10
348/348              0s 267us/step -
accuracy: 0.8125 - loss: 0.6417 - val_accuracy: 0.8621 - val_loss: 0.3762
Epoch 7/10
348/348              139s 398ms/step -
accuracy: 0.7617 - loss: 0.6600 - val_accuracy: 0.7856 - val_loss: 0.6063
Epoch 8/10
348/348              0s 234us/step -
accuracy: 0.8125 - loss: 0.5746 - val_accuracy: 0.8621 - val_loss: 1.0454
Epoch 9/10
348/348              136s 390ms/step -
accuracy: 0.7707 - loss: 0.6507 - val_accuracy: 0.7911 - val_loss: 0.5858
Epoch 10/10
348/348              0s 266us/step -
accuracy: 0.7812 - loss: 0.6545 - val_accuracy: 0.7586 - val_loss: 0.6695
```

To further enhance model performance, additional data augmentation techniques are implemented during training, introducing more variability to the dataset and helping to mitigate overfitting.

Step 10: Final Evaluation and Saving the Model

[33]:
```python
# Evaluate the model with augmented data
test_loss_augmented, test_acc_augmented = model.
 ↪evaluate(validation_generator_augmented)
print("Validation Accuracy after Augmentation: ", test_acc_augmented)

# Save the model
model.save('intel_image_classifier_augmented.h5')
```

```
87/87             14s 161ms/step -
```

```
accuracy: 0.8066 - loss: 0.5517
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or
`keras.saving.save_model(model)`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')` or `keras.saving.save_model(model,
'my_model.keras')`.
```

```
Validation Accuracy after Augmentation:  0.8029485940933228
```

Key Findings

1. The training process utilized a dataset containing 11,139 images distributed across 6 classes and a validation set with 2,781 images.

2. The evaluation results showed that the model's validation accuracy improved significantly after incorporating data augmentation techniques, which enhanced the model's robustness and capability to handle variability in image data.

Final Interpretation

The Intel Image Classification model was trained on a dataset of 11,139 images spanning 6 classes, with an additional 2,781 images used for validation. Through the training process, the model demonstrated a notable performance, achieving a validation accuracy of approximately 80.29% after employing data augmentation techniques. This indicates that the model effectively learned to generalize from the training data, improving its ability to classify images in unseen validation scenarios. The incorporation of augmented data played a crucial role in enhancing the model's robustness, making it more adept at handling variations in image inputs. Overall, the results suggest a successful implementation of a deep learning approach for image classification tasks.