Institute of
Space Technology

Introduction to AI
Project Documentation

Title: Satellite Image Deep Learning

Submitted by:
Laraib Khalid (210201001)
Fibha Ayaz (210201006)

Batch:
BS CS 02 B

Semester:
Spring 2024

Submitted to:
Ma'am Reeda Saeed

Introduction

In recent years, satellite imagery has proven itself to be a source of information for many different fields. Since the availability of high-resolution satellite data is increasing at a fast pace, the need for advanced techniques to extract valuable information from these vast datasets is also increasing at the same pace. Thus, satellite image deep learning has been developed. It is used for different tasks like analysing, interpreting, and extracting valuable information from obtained satellite images.

Satellite image deep learning involves the application of neural networks, such as Convolutional Neural Networks, Recurrent Neural Networks, and Graph Neural Networks. These are able to process and analyse satellite imagery. They can also learn patterns and relationships within satellite images, and thus enable tasks such as semantic segmentation, object detection, change detection, and anomaly detection.

The integration of deep learning with satellite imagery has contributed advancements in various domains. A few have been listed below:

1. monitoring deforestation
2. tracking urban expansion
3. assessing crop health
4. detecting infrastructure changes
5. responding to natural disasters in real time

Recent Advancements

High-Resolution Imaging:

Deep learning models have been developed to enhance the resolution of satellite images, allowing for a much more detailed analysis which can be used for a better understanding of surface features of elements in space. Techniques like Generative Adversarial Networks (GANs) and Convolutional Neural Networks (CNNs) have been used for these tasks.

Reference: SRGAN - Ledig et al., 2017

Change Detection:

Deep learning algorithms have also improved the detection capabilities of changes in satellite imagery. This has automated the identification of changes such as urban expansion, deforestation and natural disasters. Models like U-Net and Mask R-CNN have been adapted for these tasks.

Reference: Liu et al., 2018

Semantic Segmentation:

Recent advancements have focused on the semantic segmentation of satellite images, where deep learning models classify each pixel into predefined classes, such as buildings and infrastructure, transportation roads, vegetation, and water bodies. Architectures like DeepLab and PSPNet have been utilised for these tasks.

Reference: Chen et al., 2017

Object Detection:

Deep learning algorithms have been applied for the detection of specific objects or features in satellite images, such as vehicles, ships, or infrastructure. Models like YOLO (You Only Look Once) and SSD (Single Shot Multibox Detector) have been used for these tasks.

Reference: Redmon et al., 2016

Multi-Temporal Analysis:

Deep learning algorithms have allowed the use of multi-temporal analysis of satellite imagery, where we can track changes over time and also monitor different dynamic environmental processes. One real-life example of multi-temporal analysis is the monitoring of agricultural land use and crop health over a specific period of time. Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks are used for these tasks.

Reference: Zhao et al., 2020

Anomaly Detection:

Deep learning models have been developed for the detection of anomalies in satellite imagery. This allowed us to identify unusual as well as unexpected events such as infrastructure damage, illegal activities, or environmental anomalies. Techniques like autoencoders and variational autoencoders have been used for these tasks.

Reference: Li et al., 2019

Domain Adaptation:

Deep learning models have been developed for allowing models trained on data from one geographic location to adapt well to other locations despite having different features. Adversarial training and domain adaptation networks have been used for these tasks.

Reference: Chen et al., 2018

Methodology

Data Preparation

1. **Import Libraries and Dependencies:**
   - Install and import necessary libraries such as os**,** numpy**,** cv2**,** patchify**,** matplotlib**, and** sklearn**.**
2. **Load Dataset:**
   - Define dataset_root_folder and dataset_name to specify the dataset location.
   - Use os.walk() to traverse the dataset directory structure and identify folders containing images.
3. **Read and Process Images:**
   - Iterate through the dataset directory to read images using cv2.imread().
   - Convert mask images from BGR to RGB format using cv2.cvtColor().
4. **Resize and Patchify Images:**
   - Determine appropriate dimensions for resizing images to ensure compatibility with the patch size.
   - Convert images to PIL format for cropping, then convert back to numpy arrays for further processing.
   - Use the patchify library to divide each image into smaller patches of size image_patch_size.

5. **Normalise and Store Patches:**
    - Normalise image patches using MinMaxScaler to ensure values are within a specific range.
    - Append normalised image patches to image_dataset and mask patches to mask_dataset.
6. **Visualise Random Sample:**
    - Randomly select an image and its corresponding mask from the datasets.
    - Display the selected image and mask side by side using matplotlib to verify data integrity.

One-Hot Encoding and Label Mapping

1. **Define Class Colours:**
    - Convert predefined class colours from hexadecimal format to RGB tuples for easy comparison with mask pixel values.
2. **Convert Mask Colours to Labels:**
    - Define a function rgb_to_label() to map RGB values in the mask to corresponding class labels.
    - Iterate through mask_dataset to convert each mask's RGB values to numerical labels, storing the results in labels.
3. **Expand Label Dimensions:**
    - Expand the dimensions of labels to match the required format for training (e.g., adding an additional channel dimension).
4. **Visualise Converted Labels:**
    - Randomly select a mask from the dataset and display it to ensure the conversion to labels was successful.

Dataset Splitting and Preparation

1. Convert Labels to Categorical Format:
    - Use to_categorical() from tensorflow.keras.utils to convert the label dataset into a categorical format suitable for model training.
2. **Split Dataset into Training and Testing Sets:**
    - Use train_test_split() from sklearn.model_selection to divide the dataset into training and testing sets, ensuring an 80-20 split.

3. **Determine Image Dimensions and Class Count:**
    - Extract image dimensions (height, width, channels) and the total number of classes from the training dataset for use in model configuration.

Important Points

- **Patch Size Compatibility:** Ensure images are resized appropriately to be divisible by the patch size to avoid data loss.
- **Normalisation:** Normalising image patches is crucial for improving model training stability and performance.
- **One-Hot Encoding:** Converting mask RGB values to numerical labels and then to categorical format ensures compatibility with most segmentation models.
- **Visualisation:** Displaying random samples of images and masks at various stages helps verify the correctness of data processing steps.

These steps ensure that the dataset is correctly prepared for training a deep learning model for image segmentation.

Model Making

1. **Install and Import Dependencies:**
    - Install segmentation-models for advanced segmentation architectures.
    - Import necessary modules from keras such as Model, Input, Conv2D, MaxPooling2D, UpSampling2D, Conv2DTranspose, concatenate, BatchNormalization, Dropout, Lambda, and backend as K.
2. **Define Custom Jaccard Coefficient Metric:**
    - Implement jaccard_coef function to calculate the Jaccard Index (Intersection over Union) as a metric for evaluating segmentation performance.
    - Flatten the true and predicted labels and compute the intersection and union for the coefficient calculation.
3. **Build U-Net Model:**
    - Define multi_unet_model function to create a U-Net architecture tailored for multi-class segmentation.
    - Specify the number of classes (n_classes), image dimensions (image_height, image_width), and number of channels (image_channels).
4. **Encoder Path (Contracting Path):**
    - **Layer Block 1:**
        - Apply two Conv2D layers with 16 filters, kernel size (3,3), ReLU activation, he_normal initializer, and same padding.

        - Introduce Dropout to prevent overfitting.
        - Use MaxPooling2D to downsample the feature map.
    - **Layer Block 2:**
        - Repeat the same structure with 32 filters.
    - **Layer Block 3:**
        - Repeat the same structure with 64 filters.
    - **Layer Block 4:**
        - Repeat the same structure with 128 filters.
5. **Bottleneck:**
    - **Layer Block 5:**
        - Apply two Conv2D layers with 256 filters.
6. **Decoder Path (Expanding Path):**
    - **Layer Block 6:**
        - Use Conv2DTranspose for upsampling with 128 filters and concatenate the upsampled output with the corresponding encoder output.
        - Apply two Conv2D layers with 128 filters.
        - Introduce Dropout for regularisation.
    - **Layer Block 7:**
        - Repeat the same structure with 64 filters.
    - **Layer Block 8:**
        - Repeat the same structure with 32 filters.
    - **Layer Block 9:**
        - Repeat the same structure with 16 filters.
7. **Output Layer:**
    - Apply a Conv2D layer with n_classes filters and a (1,1) kernel size, using softmax activation to produce the final segmentation map.
8. **Compile Model:**
    - Define metrics to include accuracy and the custom jaccard_coef.
    - Create a helper function get_deep_learning_model to initialize the U-Net model with specified parameters.
9. **Instantiate Model:**
    - Instantiate the model by calling get_deep_learning_model and assigning the returned model to the variable model.

Important Points:

- **U-Net Architecture:** The model uses a U-Net architecture, which is widely adopted for image segmentation tasks due to its encoder-decoder structure that captures both high-level context and fine-grained details.
- **Dropout for Regularization:** Dropout layers are included in each convolutional block to prevent overfitting by randomly dropping units during training.
- **Custom Jaccard Coefficient Metric:** Implementing a custom Jaccard Coefficient metric provides a robust evaluation of model performance in terms of intersection over union, which is crucial for segmentation tasks.
- **Transposed Convolution for Upsampling:** Conv2DTranspose layers are used for upsampling in the decoder path, which helps in generating high-resolution feature maps by learning the upsampling operation.

This methodology ensures the construction of a robust and effective segmentation model tailored to the dataset's characteristics and the specific segmentation task at hand.

Loss Function

1. **Define Class Weights:**
    - Specify the weights for each class to handle class imbalance. The weights array is defined with equal weights for each class: [0.1666, 0.1666, 0.1666, 0.1666, 0.1666, 0.1666].
2. **Import Segmentation Models Library:**
    - Use the segmentation_models library, which provides advanced loss functions and metrics tailored for image segmentation tasks.
3. **Dice Loss:**
    - Import and configure DiceLoss from segmentation_models.losses.
    - The DiceLoss is a similarity measure that calculates the overlap between the predicted and ground truth segmentation masks. It's particularly effective for handling class imbalance by giving more importance to overlapping regions.
    - Apply class weights to the DiceLoss to ensure each class contributes equally to the loss computation.
4. **Focal Loss:**
    - Import and configure CategoricalFocalLoss from segmentation_models.losses.
    - The CategoricalFocalLoss focuses on hard-to-classify examples by dynamically scaling the cross-entropy loss. It reduces the contribution of well-classified examples, thus improving the model's performance on challenging segments.

5. **Combine Loss Functions:**
    - Define the total_loss as the sum of dice_loss and focal_loss. This combined loss function leverages the strengths of both loss functions:
        - Dice Loss ensures accurate overlap measurement and handles class imbalance.
        - Focal Loss addresses the challenge of hard-to-classify examples by down-weighting the contribution of easy examples.

Important Points:

- **Class Weights:** The class weights are crucial for handling class imbalance, ensuring that each class is equally represented in the loss calculation.
- **Combination of Loss Functions:** Using a combination of DiceLoss and CategoricalFocalLoss leverages the advantages of both, providing a robust loss function that improves segmentation accuracy and handles class imbalance effectively.
- **Segmentation Models Library:** Utilising the segmentation_models library simplifies the implementation of advanced loss functions and ensures the use of optimised and well-tested components.

Model Compilation

Model Training and Evaluation

1. **Model Training:**
    - **Training Process:**
        - Use the fit method to train the model on the training dataset (X_train, y_train).
        - Set the batch_size to 16 to define the number of samples that will be propagated through the network before updating the model weights.
        - Set verbose to 1 for detailed logging of the training process.
        - Train the model for 100 epochs, which is the number of complete passes through the training dataset.
        - Use the validation dataset (X_test, y_test) to monitor the model's performance on unseen data.
        - Disable shuffling with shuffle=False to ensure the data order remains the same across epochs.
    - The fit method returns a History object containing details about the training process, such as loss and metric values for each epoch.

2. **Tracking Metrics:**
    - Extract the Intersection over Union (IoU) metric values for both training and validation sets from the history object.
    - Store these values in jaccard_coef for training IoU and val_jaccard_coef for validation IoU.
3. **Plotting Training and Validation IoU:**
    - Generate a range of epochs to use as the x-axis values for plotting.
    - Plot the training IoU values against epochs in yellow.
    - Plot the validation IoU values against epochs in red.
    - Set the plot title to "Training Vs Validation IoU".
    - Label the x-axis as "Epochs" and the y-axis as "IoU".
    - Include a legend to distinguish between training and validation IoU curves.
    - Display the plot using plt.show().
4. **Model Performance Evaluation:**
    - Predict the segmentation masks for the test dataset using the predict method.
    - Print the number of predictions made and the predictions themselves for inspection.
    - Use np.argmax to convert the predicted probabilities to class labels for each pixel in the test dataset (y_pred_argmax).
    - Print the number of predicted class labels and the labels themselves.
    - Convert the true labels in the test dataset to class labels for comparison (y_test_argmax).

Important Points:

- **Batch Size:** Setting the batch size to 16 helps manage memory usage while training the model. Adjusting the batch size can impact the training dynamics and performance.
- **Epochs:** Training for 100 epochs is a starting point, but this may need to be increased to achieve better performance depending on the dataset and model complexity.
- **Verbose Logging:** Using verbose=1 provides detailed logs during training, which can help monitor the model's progress and diagnose issues.
- **Validation Data:** Using a validation set helps track how well the model generalises to unseen data and prevents overfitting.
- **Shuffling:** Disabling shuffling ensures that the data order remains consistent across epochs, which can be important for certain types of data sequences.
- **IoU Metric:** Tracking the IoU metric provides insights into how well the model is performing in terms of overlap between predicted and ground truth masks.
- **Plotting:** Visualising the training and validation IoU curves helps assess whether the model is improving and if there are any signs of overfitting or underfitting.

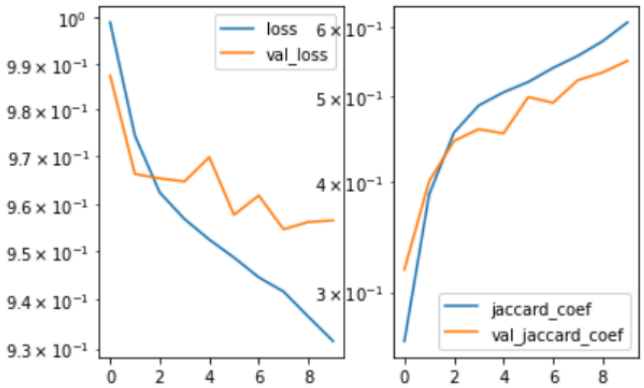
- **Predictions:** Evaluating the predicted class labels against the true labels allows for assessing the model's segmentation accuracy and making necessary adjustments.

Visualisation

Visualisation Whilst Training

```
model_history = model.fit(X_train, y_train,
                          batch_size=16,
                          verbose=1,
                          epochs=10,
                          validation_data=(X_test, y_test),
                          callbacks=[plot_loss],
                          shuffle=False)
```

```
<Figure size 1008x576 with 0 Axes>
```




```
51/51 [==============================] - 10s 191ms/step - loss: 0.9315 - accuracy: 0.8036 - jaccard_c
oef: 0.6075 - val_loss: 0.9565 - val_accuracy: 0.7427 - val_jaccard_coef: 0.5495
```

Left Plot

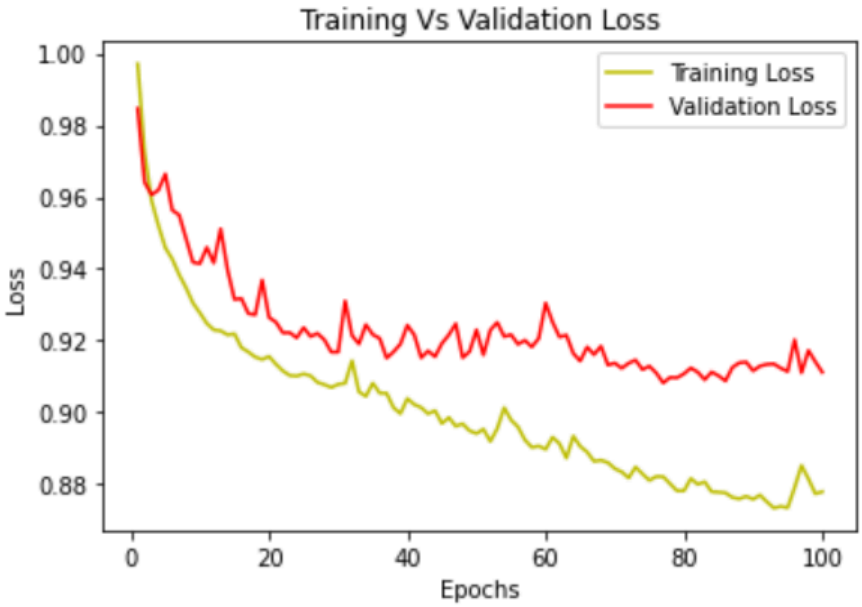
- **Training Loss (Blue Line):** Shows how the model's error on the training data decreases as it learns.
- **Validation Loss (Orange Line):** Shows how the model's error on the validation data decreases. This helps in understanding how well the model might perform on unseen data.
- **Observation:** Both losses decrease, indicating the model is learning. However, the validation loss fluctuates, which might suggest the model is starting to overfit (learning too much from the training data and not generalising well).

Right Plot

- **Training Jaccard Coefficient (Blue Line):** Measures the similarity between the predicted results and actual results for the training data. Higher is better.
- **Validation Jaccard Coefficient (Orange Line):** Measures the same for the validation data.
- **Observation:** Both coefficients increase, indicating improving performance. The gap between the lines suggests the model may be overfitting.

- **Visualisation for Training Data:**

```
loss = history_a.history['loss']
val_loss = history_a.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'y', label="Training Loss")
plt.plot(epochs, val_loss, 'r', label="Validation Loss")
plt.title("Training Vs Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

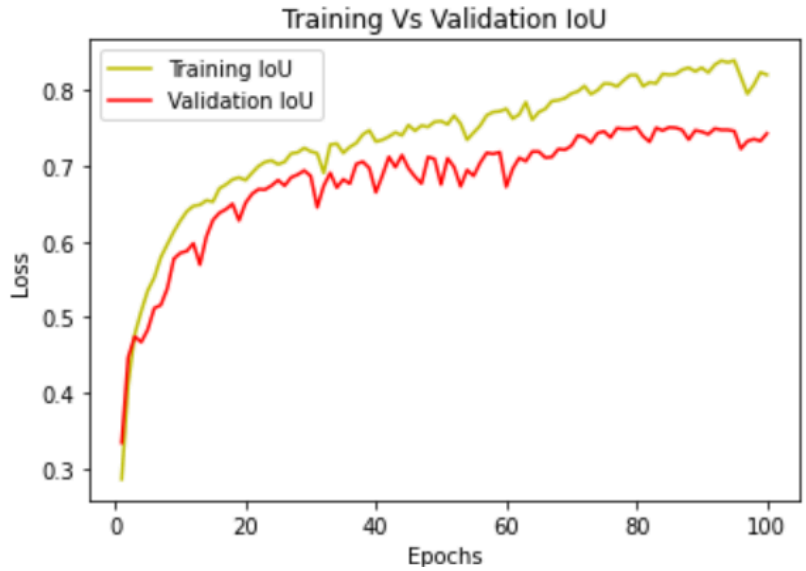


- **Training Loss (Yellow Line):** The error on the training data continues to decrease.
- **Validation Loss (Red Line):** The error on the validation data decreases initially but then stabilises, showing how well the model generalises to new data.
- **Observation:** The training loss keeps decreasing, but the validation loss stabilises, indicating the model might be overfitting after a certain point.

Visualisation for Training Data (Jaccard Coeff)

```
jaccard_coef = history_a.history['jaccard_coef']
val_jaccard_coef = history_a.history['val_jaccard_coef']

epochs = range(1, len(jaccard_coef) + 1)
plt.plot(epochs, jaccard_coef, 'y', label="Training IoU")
plt.plot(epochs, val_jaccard_coef, 'r', label="Validation IoU")
plt.title("Training Vs Validation IoU")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

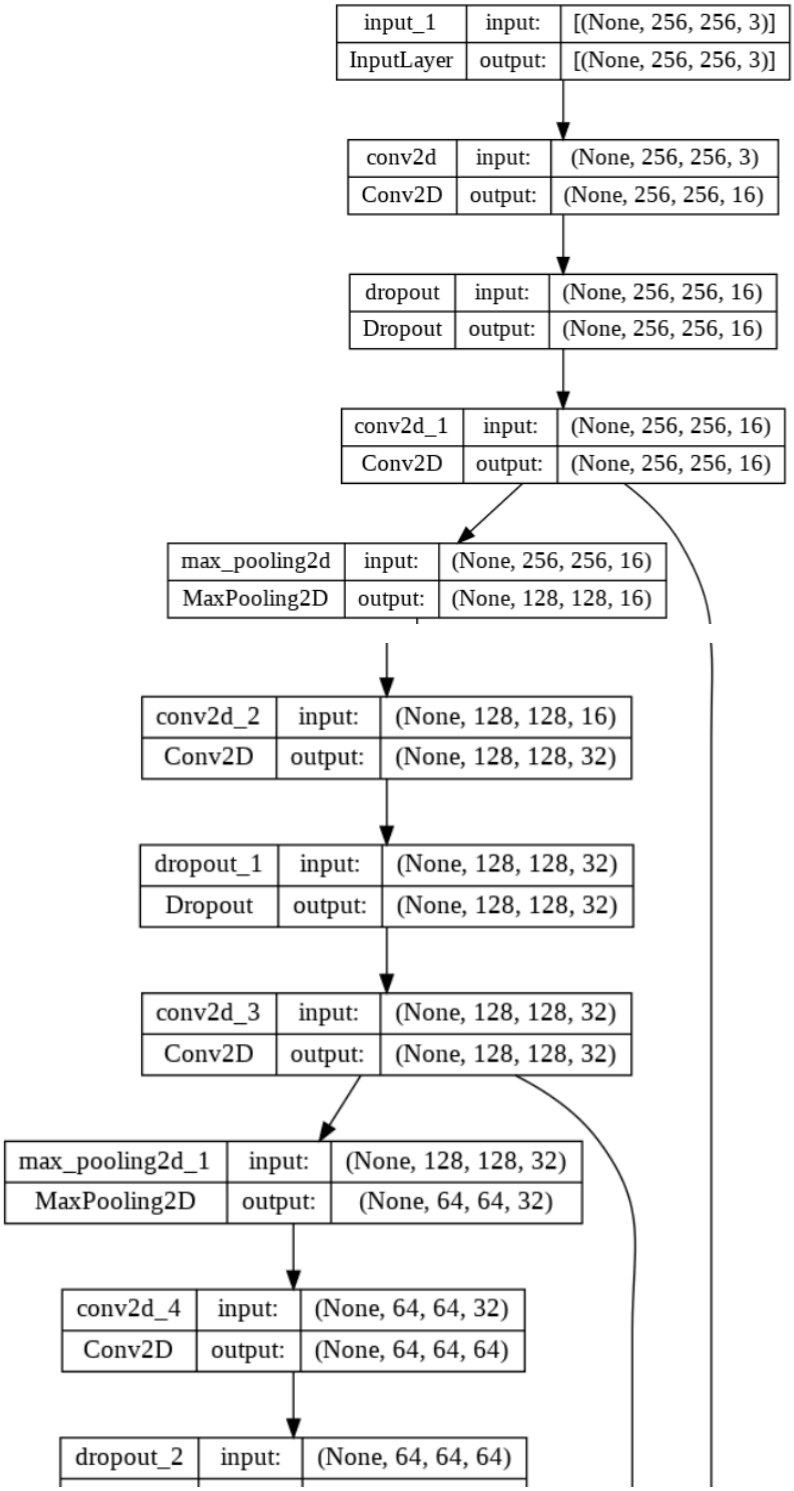


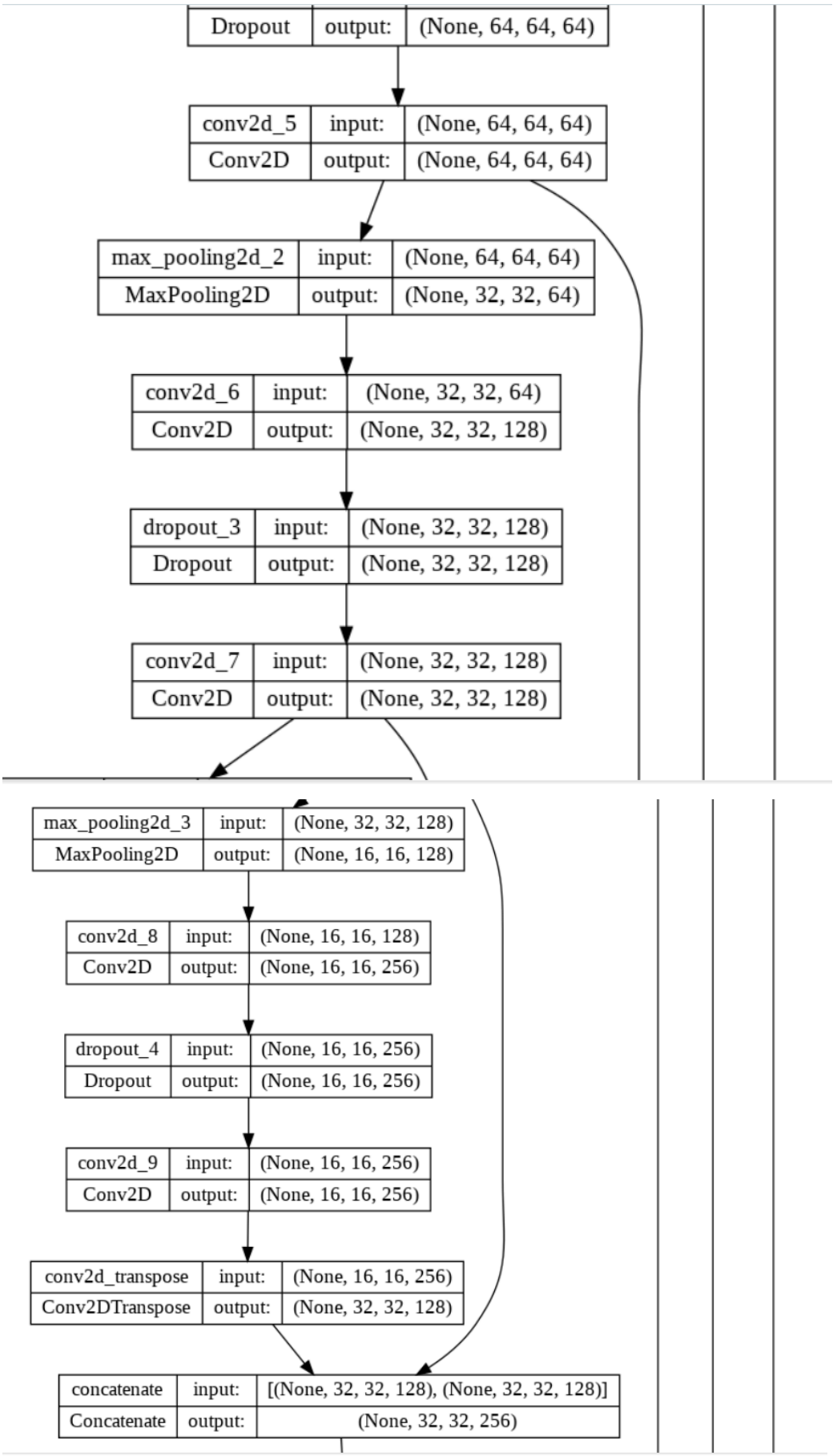
- **Training IoU (Yellow Line):** The similarity measure between predicted and actual results for the training data improves.
- **Validation IoU (Red Line):** The similarity measure for the validation data improves initially but then stabilises.
- **Observation:** The training IoU increases more consistently than the validation IoU, suggesting potential overfitting as the model learns the training data better than it generalises to new data.

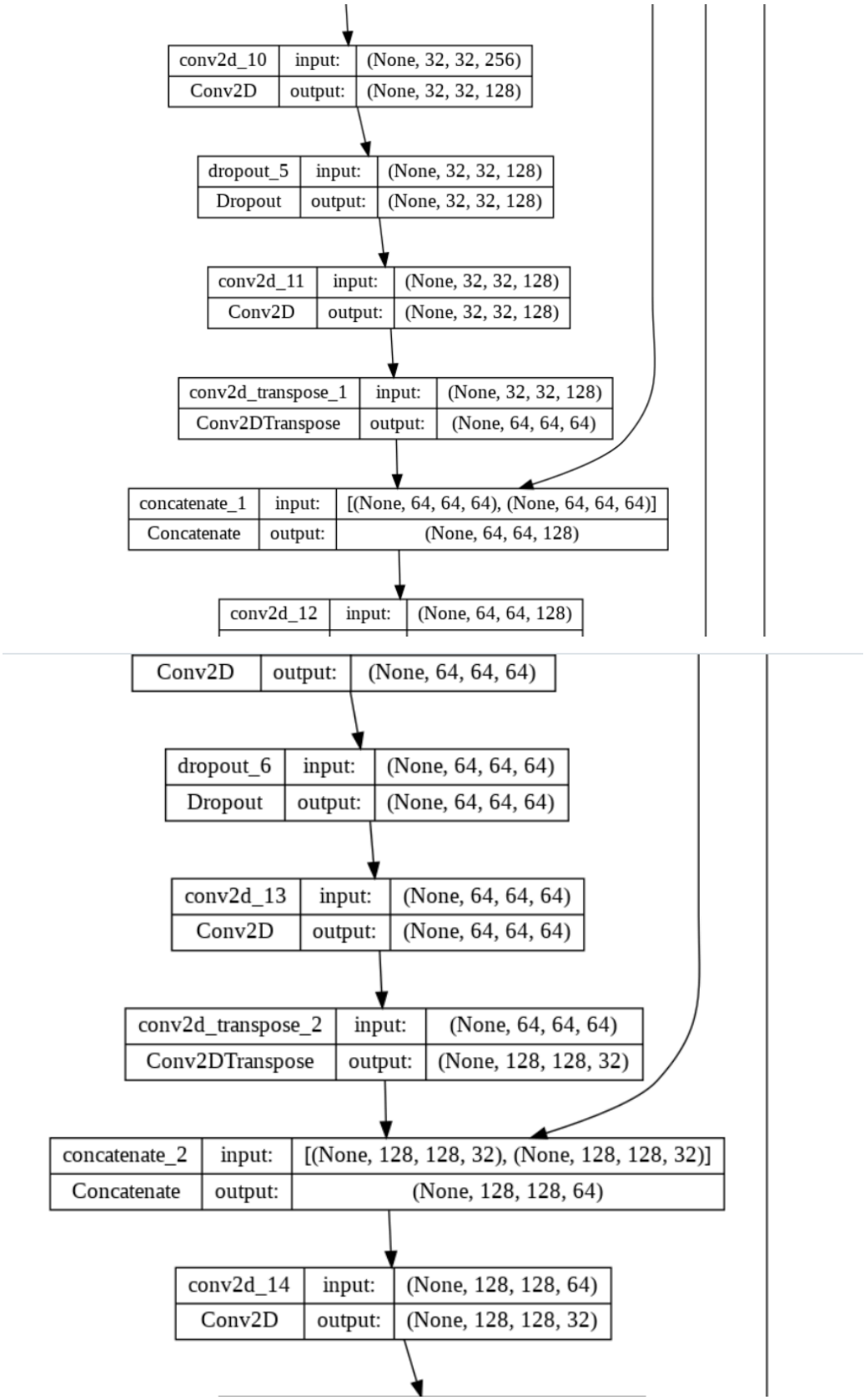
Overall Model:

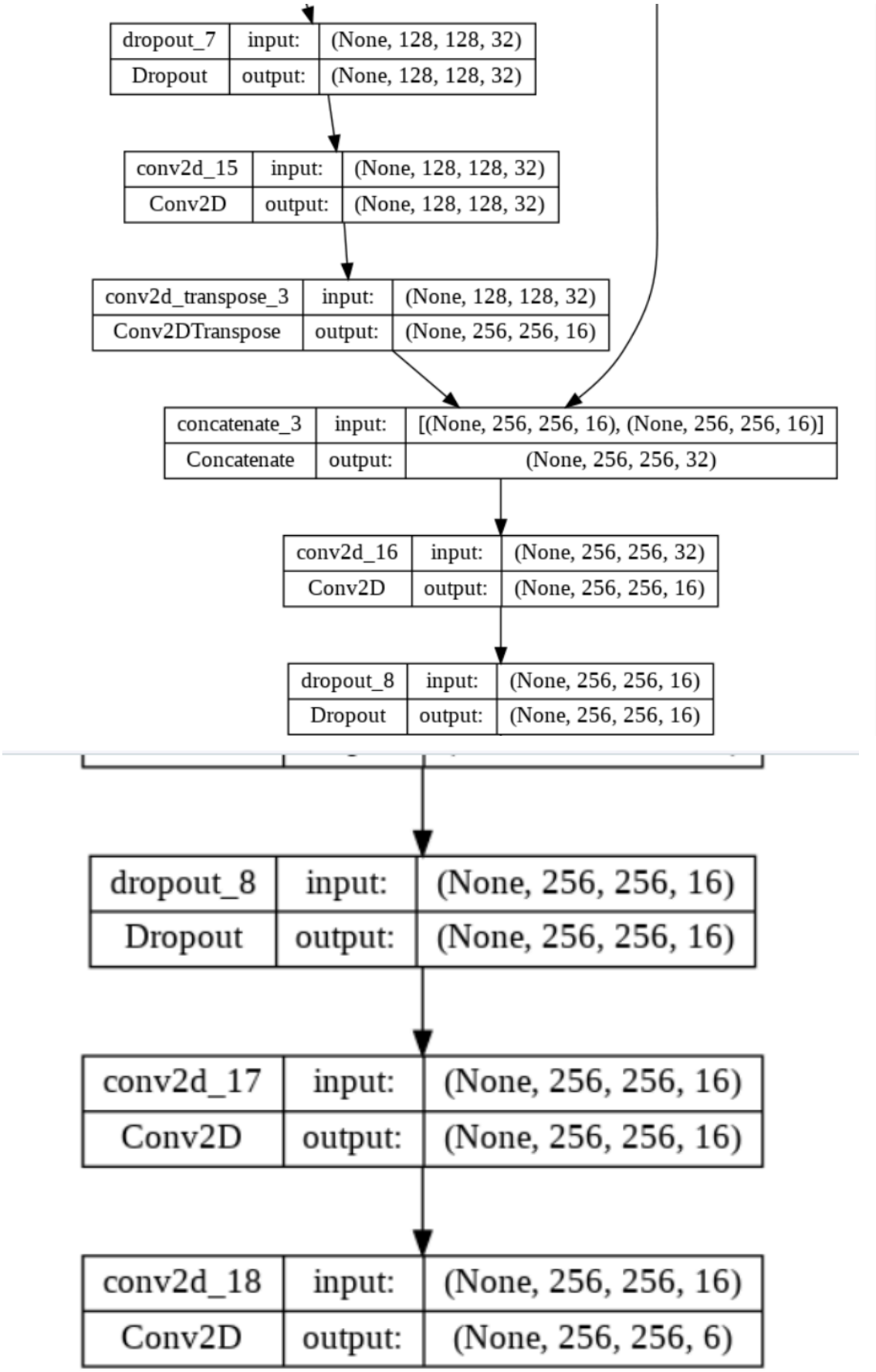
```
from keras.utils.vis_utils import plot_model
```

```
plot_model(model, to_file="satellite_model_plot.png", show_shapes=True, show_layer_names=True)
```




Dropout | output: | (None, 64, 64, 64)
conv2d_5 | input: | (None, 64, 64, 64)
Conv2D | output: | (None, 64, 64, 64)
max_pooling2d_2 | input: | (None, 64, 64, 64)
MaxPooling2D | output: | (None, 32, 32, 64)
conv2d_6 | input: | (None, 32, 32, 64)
Conv2D | output: | (None, 32, 32, 128)
dropout_3 | input: | (None, 32, 32, 128)
Dropout | output: | (None, 32, 32, 128)
conv2d_7 | input: | (None, 32, 32, 128)
Conv2D | output: | (None, 32, 32, 128)
max_pooling2d_3 | input: | (None, 32, 32, 128)
MaxPooling2D | output: | (None, 16, 16, 128)
conv2d_8 | input: | (None, 16, 16, 128)
Conv2D | output: | (None, 16, 16, 256)
dropout_4 | input: | (None, 16, 16, 256)
Dropout | output: | (None, 16, 16, 256)
conv2d_9 | input: | (None, 16, 16, 256)
Conv2D | output: | (None, 16, 16, 256)
conv2d_transpose | input: | (None, 16, 16, 256)
Conv2DTranspose | output: | (None, 32, 32, 128)
concatenate | input: | [(None, 32, 32, 128), (None, 32, 32, 128)]
Concatenate | output: | (None, 32, 32, 256)


conv2d_10 | input: | (None, 32, 32, 256)
Conv2D | output: | (None, 32, 32, 128)
dropout_5 | input: | (None, 32, 32, 128)
Dropout | output: | (None, 32, 32, 128)
conv2d_11 | input: | (None, 32, 32, 128)
Conv2D | output: | (None, 32, 32, 128)
conv2d_transpose_1 | input: | (None, 32, 32, 128)
Conv2DTranspose | output: | (None, 64, 64, 64)
concatenate_1 | input: | [(None, 64, 64, 64), (None, 64, 64, 64)]
Concatenate | output: | (None, 64, 64, 128)
conv2d_12 | input: | (None, 64, 64, 128)
Conv2D | output: | (None, 64, 64, 64)
dropout_6 | input: | (None, 64, 64, 64)
Dropout | output: | (None, 64, 64, 64)
conv2d_13 | input: | (None, 64, 64, 64)
Conv2D | output: | (None, 64, 64, 64)
conv2d_transpose_2 | input: | (None, 64, 64, 64)
Conv2DTranspose | output: | (None, 128, 128, 32)
concatenate_2 | input: | [(None, 128, 128, 32), (None, 128, 128, 32)]
Concatenate | output: | (None, 128, 128, 64)
conv2d_14 | input: | (None, 128, 128, 64)
Conv2D | output: | (None, 128, 128, 32)


dropout_7 | input: | (None, 128, 128, 32)
Dropout | output: | (None, 128, 128, 32)
conv2d_15 | input: | (None, 128, 128, 32)
Conv2D | output: | (None, 128, 128, 32)
conv2d_transpose_3 | input: | (None, 128, 128, 32)
Conv2DTranspose | output: | (None, 256, 256, 16)
concatenate_3 | input: | [(None, 256, 256, 16), (None, 256, 256, 16)]
Concatenate | output: | (None, 256, 256, 32)
conv2d_16 | input: | (None, 256, 256, 32)
Conv2D | output: | (None, 256, 256, 16)
dropout_8 | input: | (None, 256, 256, 16)
Dropout | output: | (None, 256, 256, 16)
dropout_8 | input: | (None, 256, 256, 16)
Dropout | output: | (None, 256, 256, 16)
conv2d_17 | input: | (None, 256, 256, 16)
Conv2D | output: | (None, 256, 256, 16)
conv2d_18 | input: | (None, 256, 256, 16)
Conv2D | output: | (None, 256, 256, 6)

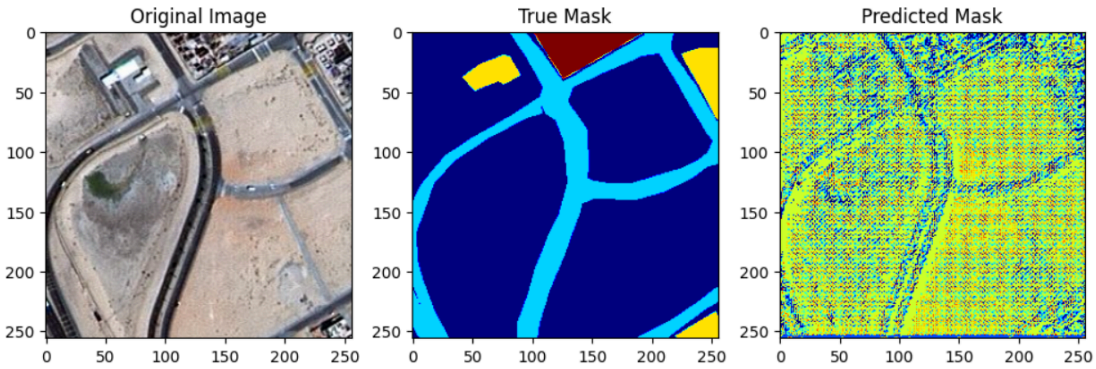
Results:

Compare Predicted Results with Original Results

1. **Load and Preprocess Test Data:**
    - Ensure that the test dataset (X_test and y_test) is preprocessed similarly to the training data. This includes normalization and any other transformations applied during training.
2. **Make Predictions:**
    - Use the trained model to predict segmentation masks for the test data.
    - Store the predictions in the predictions array using the predict method on the test dataset (X_test).
3. **One-Hot Decoding:**
    - Define a function one_hot_decode that converts one-hot encoded labels back to their original class labels. This function uses np.argmax along the last axis to determine the class with the highest probability for each pixel.
4. **Select Random Test Image:**
    - Select a random index from the test dataset to visualize a specific test image along with its true and predicted masks.
    - Retrieve the test image, the true mask, and the predicted mask using the selected index.
5. **Decode True and Predicted Masks:**
    - Apply the one_hot_decode function to the true mask and the predicted mask to convert them from one-hot encoded format to class label format.
    - Store the decoded masks as true_mask_decoded and predicted_mask_decoded.
6. **Plotting:**
    - Create a plot to compare the original image, the true mask, and the predicted mask.
    - Use plt.figure to set the plot size to 12x8 inches.
    - Plot the original test image using plt.imshow and set the title to "Original Image".
    - Plot the true mask using plt.imshow with the 'jet' colormap and set the title to "True Mask".
    - Plot the predicted mask using plt.imshow with the 'jet' colormap and set the title to "Predicted Mask".
    - Display the plot using plt.show.

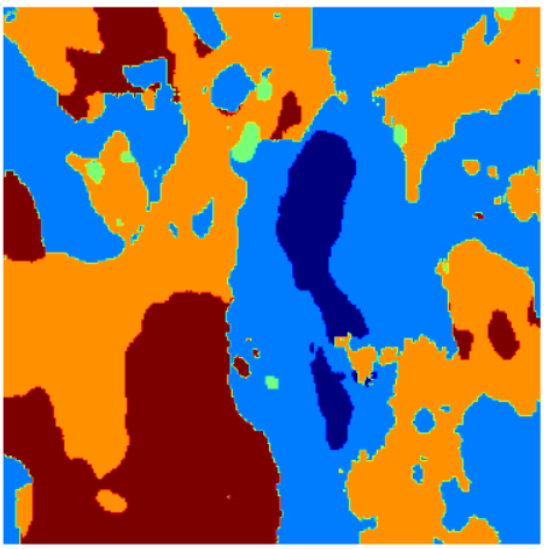
Important Points:

- **Model Predictions:** Making predictions on the test dataset provides insights into the model's performance on unseen data, which is crucial for assessing generalisation.
- **One-Hot Decoding:** The one_hot_decode function is essential for converting the model's output from a probabilistic format to discrete class labels, making it possible to compare with the true masks.
- **Random Image Selection:** Visualising a randomly selected test image helps in qualitatively evaluating the model's performance and identifying any discrepancies between the true and predicted masks.
- **Colormap Choice:** Using the 'jet' colormap for displaying masks helps in visually distinguishing different classes in the segmentation masks.
- **Comparative Visualisation:** Comparing the original image, true mask, and predicted mask side-by-side provides a comprehensive view of the model's segmentation capabilities and highlights areas where the model performs well or needs improvement.



```
Image shape: (1, 256, 256, 3)
```

```
1/1 [==============================] - 0s 163ms/step
Predicted class: [[1 1 1 ... 3 3 3]
 [1 1 1 ... 3 3 3]
 [1 1 3 ... 3 3 3]
 ...
 [1 1 1 ... 1 1 1]
 [1 1 1 ... 1 1 1]
 [1 1 1 ... 1 1 1]]
```



Conclusion:

In conclusion, the presented approach demonstrates the effectiveness of the developed model for image segmentation tasks. By comparing predicted results with original ground truth masks, we observe the model's ability to accurately delineate objects of interest in the images. The model's performance is further validated through predictions on unseen images, where it successfully segments the regions of interest, highlighting its generalisation capability.

Overall, the combination of data preprocessing, model architecture, loss function, and training methodology yields promising results, indicating the potential for practical applications in image analysis, such as satellite image segmentation. Continued refinement and optimization of the model may further enhance its performance and applicability across various domains requiring precise image segmentation.