Semantic segmentation of aerial imagery using a deep learning U-Net model

by The-Huan Hoang

Word Count: 2035 (excluding code blocks and references)

In [ ]:
```
#Suppress warnings
import warnings
warnings.filterwarnings("ignore")

# Importing the libraries
try :
    import tensorflow
    import imageio.v2
    import PIL
except ImportError:
    !pip install tensorflow imageio PIL
import tensorflow as tf
import imageio.v2 as imageio
from PIL import Image
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Dropout, BatchNormalization,
from tensorflow.keras.losses import SparseCategoricalCrossentropy
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os
from urllib.parse import urlparse
from requests import get
import zipfile

# Set the seed for all random operations
randomseed = 2
```

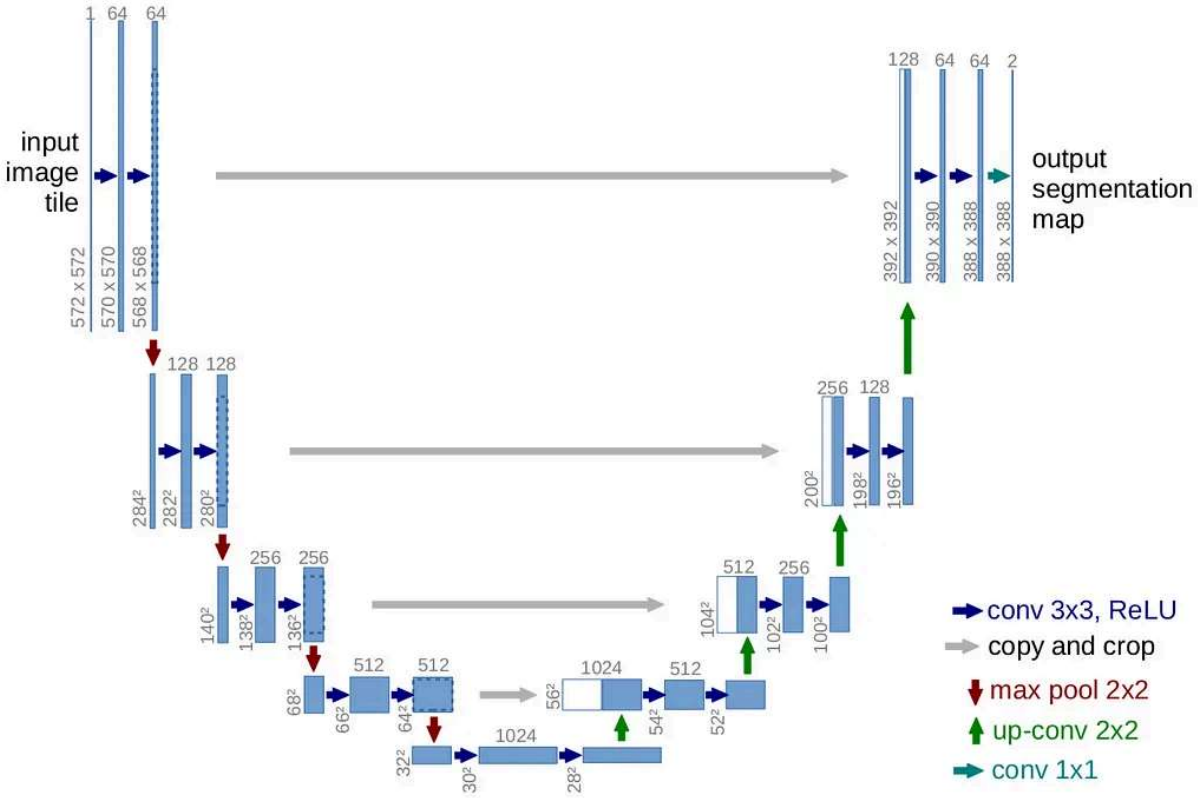
1. About Semantic Segmentation and U-Net

Semantic segmentation is computer vision task of assigning a discrete label value to each pixel in an image to describe the nature of the objects present (e.g. road, sky, people, green). It is different from other computer vision tasks, such as *Image Classification* which labels the entire image (e.g. rural, city) , or *Instance Segmentation* which distinguishes individual subjects (e.g. each car, each building), or *Object Detection* which draws bounding box around each instance. It is widely used in diverse fields such as medical, robotics, and satellite imagery analysis where an understanding of what comprise the entire scene is important.

Whereas the traditional Convolutional Neural Network can predict single output from images, making it appropriate for Image Classification, a **Fully Convolutional Network** (or FCN, an specialised CNN

architecture) can perform semantic segmentation tasks thanks to its final upsampling steps to the original image's dimension so that each pixel's label can be learned.

There is an ever growing literature body on diverse FCN architectures for specific purposes. Among them, **U-Net** has garnered popularity due to its capabilities to effectively learn spatial patterns crucial for high-resolution image classification.



1.1 What makes up a U-Net architecture?

Originally devised for used in biomedical field (Ronneberger et al., 2015), a U-Net architecture in its most basic form is made up of two main sections: the *decoders* and the *encoders*. The encoders capture features and contextual information from the input full-color image, while the decoders reconstruct those data into the output segmentation map.

In both, there are multiple blocks made up of convolutional layers, and one block feeds into the next one with *maxpooling* (for encoders to reduce the spatial resolution) or *upsampling* (for decoders to revert back to the original resolution). Its signature quasi-symmetrical 'U'-shaped architecture also helps preserve spatial relationships between the input image and the ouput segmentation map by allowing 'skip-level' learning.

This is different from other traditional FCN architectures where the upsampling is done only at the end, therefore not allowing for accurate pixel-based label learning.

1.2 Training a U-Net architecture

A U-net can be improved by employing a pre-trained backbones so that the model does not have to learn from scratch. We could also experiment with different model parameters and tune various training hyperparameters.

While tuning hyperparameters such as epoch and batch size is relatively universal, tuning model parameters presents a more unique opportunity to harness the power of U-Net. Lee et al. (2022) in their work to improve U-Net for land cover classification specifically explored tuning different model parameters such as:

- **Input Image size**
- **Model Depth**, i.e. how many layers of encoder/decoder blocks
- **Block Width**, i.e. how many convolutional layers to include in each encoder/decoder block
- **Kernel Size**, i.e. the size of the filter used in the convolution operations

Moreover, Wang et al. (2023), in their research on high-resolution remote sensing images semantic segmentation with Unet and SegNet, observed better remedy for the vanishing gradient problem when including a Batch Normalisation each the end of each block and using the **ELU** (Exponential Linear Unit) activation function instead of the more popularly used **ReLU** (Rectified Linear Unit).

1.3 Project Scope

Having understood the relevance of U-Net for semantic segmentation tasks, we will construct the model based on the architecture prescribed in (Ronneberger et al., 2015). Then, we will tune the relevant parameters in order to train the model to perform semantic segmentation tasks on a data set of drone aerial imagery. Lastly, we will assess its accuracy metrics and determine opportunities for future improvement. Due to the limited scope of this project, the chosen parameters for experimentation are:

1. Block Width: 2, 4 or 6 convolutional layers per block
2. Kernel Size: 3x3 or 5x5
3. Activation function: ReLU or ELU

2. About the dataset

For this segmentation task, we will use a subset of the Urban Drone Dataset introduced in Chen et al. (2018), and mader available on the authors' GitHub.

This is a collection of drone image dataset collected at Peking University, Huludao city, Henan University and Cangzhou city. There are about 150 images and their annotated ground truth masks provided as follows:

Class	Label
Other	0
Facade	1
Road	2
Vegetation	3
Vehicle	4
Roof	5

Lastly, although the provided dataset has additionally been split into 'train' and 'val' subsets, for more flexibility in train/test splitting later in the process, they have also been rejoined into one and hosted on

Dropbox.

3. Loading and preprocessing the dataset

First, we will define some helper functions to download the dataset from remote, load the images and annotations, and preprocess the images to the configuration appropriate for the U-Net model. For data pre-processing, we essentially resize all source and annotated images into a uniform dimension, and then reshape it into arrays with height, width, and channels. For source images, there are 3 channels representing each of the RGB values, while for annotations there is one channel that contains the integer value of the pixel.

In [ ]:
```python
def cache_data(src: str, dest: str) -> str:
    url = urlparse(src)  # We assume that this is some kind of valid URL
    fn = os.path.split(url.path)[-1]  # Extract the filename
    dfn = os.path.join(dest, fn)  # Destination filename as path

    if not os.path.isfile(dfn):
        print(f"{dfn} not found, downloading!")
        path = os.path.split(dest)
        if len(path) >= 1 and path[0] != "":
            os.makedirs(os.path.join(*path), exist_ok=True)
        with open(dfn, "wb") as file:
            response = get(src)
            file.write(response.content)
        print("\tDone downloading...")
    else:
        print(f"Found {dfn} locally!")
    return dfn
```

In [ ]:
```python
def LoadImg (path1, path2):
    # Read the images folder like a list
    img_dataset = os.listdir(path1)
    ann_dataset = os.listdir(path2)

    # Make a list for images and annotations filenames
    img_list = []
    ann_list = []
    for file in img_dataset:
        img_list.append(file)
    for file in ann_dataset:
        ann_list.append(file)

    # Sort the lists to get both of them in same order (the dataset has exactly the same name
    img_list.sort()
    ann_list.sort()

    return img_list, ann_list
```

In [ ]:
```python
def PreprocessImg(img_list, ann_list, target_img, target_ann, path1, path2):

    # Pull the relevant dimensions for image and annotations
    n = len(img_list)                    # number of images
    img_h,img_w,img_c = target_img   # set target dimenions and channels of image for pre-proc
    ann_h,ann_w,ann_c = target_ann   # set target dimenions and channels of annotations for p

    # Define X and y as arrays of transformed images and annotations
```

```
    X = np.zeros((n,img_h,img_w,img_c), dtype=np.float32)
    y = np.zeros((n,ann_h,ann_w,ann_c), dtype=np.int32)

    # Resize images and annotations to target dimensions
    for file in img_list:
        # convert image into an array of desired shape (3 channels)
        index = img_list.index(file)
        path = os.path.join(path1, file)
        img = Image.open(path).convert('RGB')
        img = img.resize((img_h,img_w))
        img = np.reshape(img,(img_h,img_w,img_c))
        img = img/256
        X[index] = img

        # convert annotations into an array of desired shape (1 channel)
        ann_index = ann_list[index]
        path = os.path.join(path2, ann_index)
        ann = Image.open(path)
        ann = ann.resize((ann_h, ann_w))
        ann = np.reshape(ann,(ann_h,ann_w,ann_c))
        y[index] = ann

    return X, y
```

With the helper functions set up, we proceed to downloading, importing, and preprocessing the Urban Drone Dataset.

In [ ]:
```
%%time
# Download the dataset from the link (4 minutes)
url = 'https://www.dropbox.com/scl/fi/yib54tq15a7orosvfc1vb/UDD6.zip?rlkey=h3ny22z1ohqemzvjqir
dest = 'data'
dfn = cache_data(url, dest)
```

```
Found data\UDD6.zip locally!
CPU times: total: 0 ns
Wall time: 0 ns
```

In [ ]:
```
# Unzip the dataset
with zipfile.ZipFile(dfn, 'r') as zip_ref:
    zip_ref.extractall(dest)
```

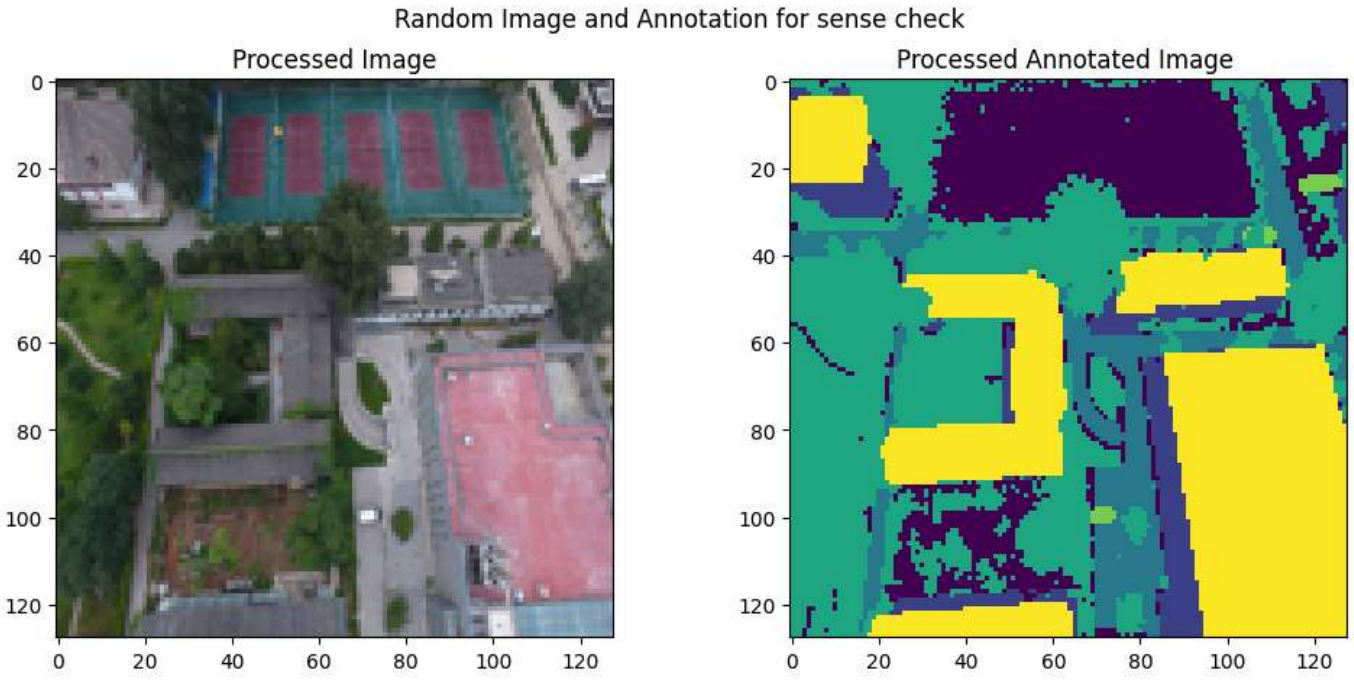
In [ ]:
```
# Load the images and annotations
path1 = os.path.join('data','UDD6','src')
path2 = os.path.join('data','UDD6','gt')
img_files, ann_files = LoadImg (path1, path2)

# Define the desired shape for UNet
target_img = [128, 128, 3]
target_ann = [128, 128, 1]

# Process data
X, y = PreprocessImg(img_files,ann_files, target_img, target_ann, path1, path2)
```

Before continuing, let's visualise the ingested dataset, both the source image and the annotated ground truth to make sure everything has been done correctly.

In [ ]:
```
# Visualize the output at random
image_index = np.random.randint(0, X.shape[0])
```

```
fig, ax = plt.subplots(1, 2, figsize=(12, 5))
ax[0].imshow(X[image_index])
ax[0].set_title('Processed Image')
ax[1].imshow(y[image_index,:,:,0])
ax[1].set_title('Processed Annotated Image ')
fig.suptitle('Random Image and Annotation for sense check')
```

Out[ ]: Text(0.5, 0.98, 'Random Image and Annotation for sense check')

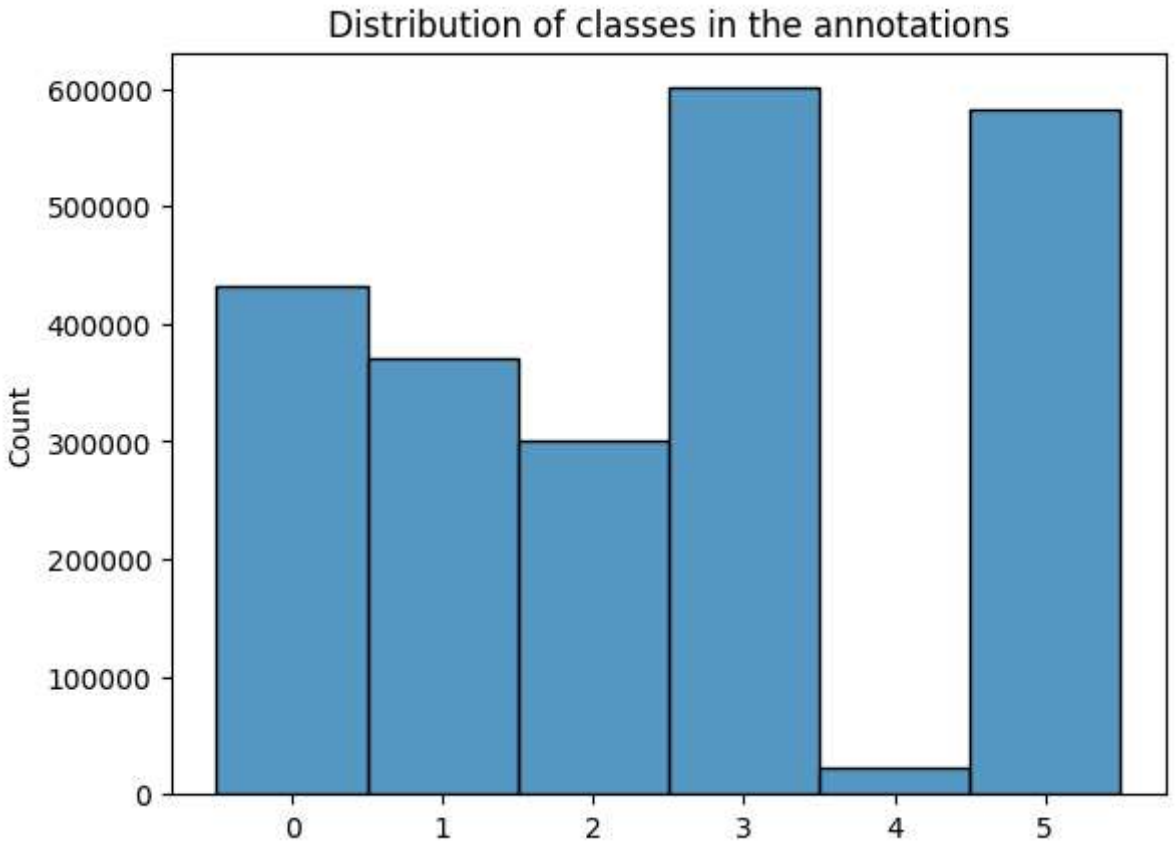


We have managed to align the source and ground truth image into corresponding pairs, while also resizing them to a dimension of 128 x 128. The annotations seem to match the source image in coverage and value, so we are generally confident that the data is of high quality.

To check if there is over-representation of one class over others in the dataset, it is worth having an idea of the distribution of the 6 classes within the entire dataset to see if there are class weight imbalances that might introduce bias in the model.

In [ ]:
```
# Visualise distribution of classes in the annotations
sns.histplot( y.flatten(), discrete=True)
plt.title('Distribution of classes in the annotations')
```

Out[ ]: Text(0.5, 1.0, 'Distribution of classes in the annotations')



Based on the histogram for label counts over the whole dataset, we can see that:

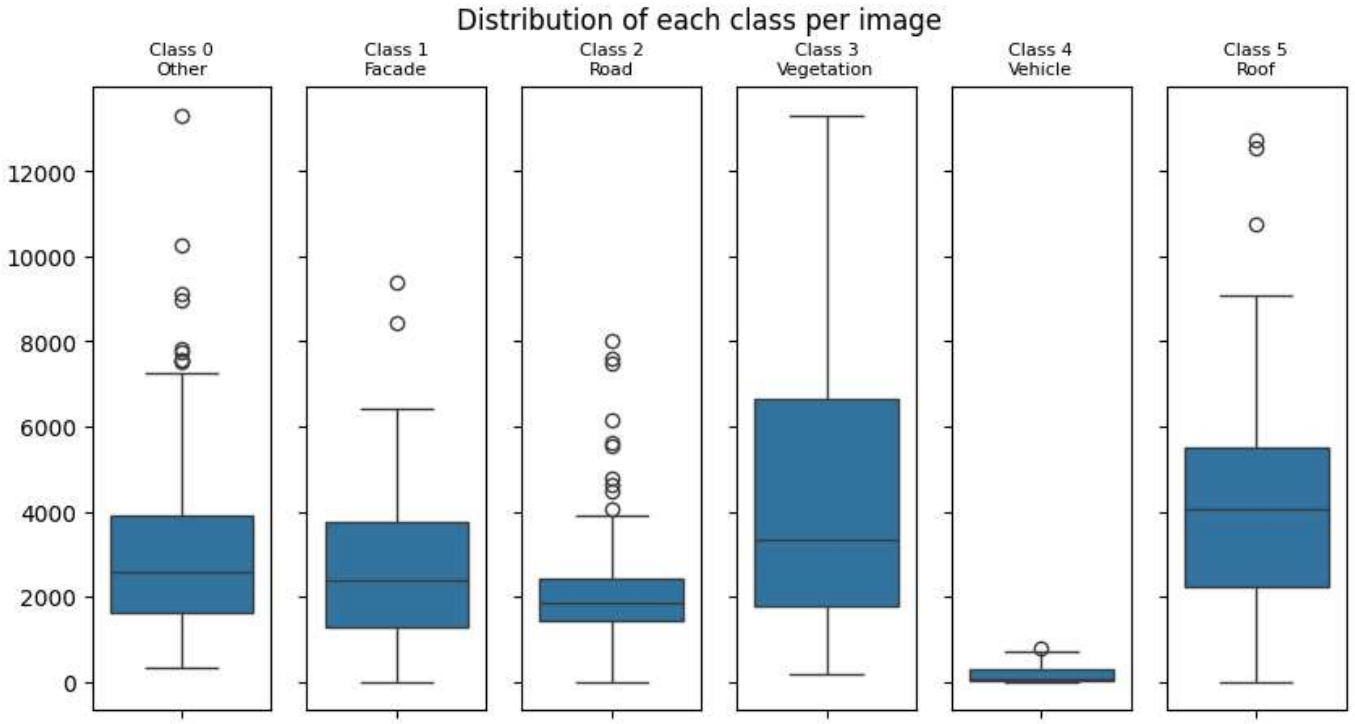
- Class 3 and 5 (Vegetation and Roof) have the highest occurences which makes sense for an aerial drone dataset.
- Class 0, 1, and 2 representing Other, Facade, and Road respectively have fewer.
- Class 4 (Vehicle) have the least occurences.

The model may have the most data to learn and predict class 3 and 5, whereas its predicting class 4 may be limited. We will refer back to this after we have trained the model.

Moreover, if we visualise the distributions of each class among the images, we gain further insights: First, the mean counts of the labels relative to one another resemble the total count reflected in the histogram. This means, it is not the case that most images have a disproportionate count of certain labels. Second, there are some extreme outliers in Class 0, 2, and 5. While it is not an inherent issue, if these are disproportionately sorted into the test dataset, leaving fewer data to train the model on, their predictions might also be affected. Considering this is a rather small dataset, such inadvertent train/test split bias could introduce inaccuracies.

In [ ]:
```
# Boxplot distribution of each class per image
classes = ['Other', 'Facade', 'Road', 'Vegetation', 'Vehicle', 'Roof']
fig, ax = plt.subplots(1, 6, figsize=(10, 5), sharey=True)
for i in range(6):
    sns.boxplot(data = np.sum(y == i, axis=(1,2,3)), ax=ax[i])
    #label each boxplot with class number and name
    ax[i].set_title(f'Class {i}\n{classes[i]}', fontsize=8)
    ax[i].set_xticklabels([''])
    ax[i].set_xlabel('')
fig.suptitle('Distribution of each class per image')
```

Out[ ]:
```
Text(0.5, 0.98, 'Distribution of each class per image')
```



For our training, we will use a 80-20 split on the dataset.

In [ ]:
```
# Split the data into training and testing sets (80-20)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=randoms

# Doublecheck array shapes
print("X-train Shape:", X_train.shape)
print("Y-train shape:", y_train.shape)
print("X-test Shape:", X_test.shape)
print("Y-test shape:", y_test.shape)
print(np.unique(y_train))
print(np.unique(y_test))
```

```
X-train Shape: (112, 128, 128, 3)
Y-train shape: (112, 128, 128, 1)
X-test Shape: (29, 128, 128, 3)
Y-test shape: (29, 128, 128, 1)
[0 1 2 3 4 5]
[0 1 2 3 4 5]
```

4. Build UNet architecture block by block

Despite its complexity, the U-Net is fundamentally a chain of blocks of different types of node layers, that feed forward to another in the prescribed U-shape. In order to build our U-Net, we will construct block-by-block the following:

- **Convolutional Block** the fundamental piece of larger blocks, consisting of 2 conv layers sandwiching a batch normalisation to improve training speed and stability
- **Encoder Block** puts the input image through convolutional layers before maxpooling to reduce the dimension. Each block returns the input for the next block, together with a skip layer input to be used in the corresponding decoder block.
- **Decoder Block** first upsamples the image from the previous layer to a bigger size before concatenating with the skip layer inputs from its corresponding encoder block. The image is then

put through convolution layers and returns what will be the input for the next decoder block.

In [ ]:
```
# Build the convolutional blocks layer
def convol_block(input,activation,n_filters, kernel):
    convol = Conv2D(n_filters,
                    kernel_size=kernel,
                    activation=activation,
                    kernel_initializer='HeNormal',
                    padding='same')(input)
    convol = BatchNormalization()(convol)
    convol = Conv2D(n_filters,
                    kernel_size=kernel,
                    activation=activation,
                    kernel_initializer='HeNormal',
                    padding='same')(convol)
    return convol
```

In [ ]:
```
# Build encoder blocks
def EncoderBlock(input, n_conv , activation, n_filters, kernel, stride, dropout_prob=0.3, max

    # Add convolution Layers
    convol_input = input
    for i in range(n_conv):
        convol_input = convol_block(convol_input, activation, n_filters, kernel)
    convol = convol_input

    # Skip level connection serves as input to the symmetrical decoder layer
    skip_block = convol

    # Dropout parameter added to prevent overfitting by randomly dropping a subset of input u
    convol = Dropout(dropout_prob)(convol)

    # Max pooling layer to reduce the spatial dimensions of the output volume for the next la
    if max_pooling:
        next_block = MaxPooling2D(pool_size = (stride,stride))(convol)
    else:
        next_block = convol

    return next_block, skip_block
```

In [ ]:
```
# Build decoder blocks
def DecoderBlock(prev_layer_input, skip_layer_input, n_conv, activation,n_filters, kernel, st

    # Transpose layer to first increase the size of the image
    upsample = Conv2DTranspose(
                 n_filters,
                 (kernel,kernel),
                 strides=(stride,stride),
                 padding='same')(prev_layer_input)

    # Merge inputs from the skip block and previous block
    merge = concatenate([upsample, skip_layer_input], axis=3)

    # Add Conv Layers with relu activation and HeNormal initialization. No batch normalizatio
    convol_input = merge
    for i in range(n_conv):
        convol_input = convol_block(convol_input, activation, n_filters, kernel)
    next_block = convol_input
```

```
    return next_block
```

We can now combine all encoder and decoder blocks in the prescribed sequence to form the **U-Net architecture**. Note that since model depth is not our intended parameter to tune, the architecture is built with a fixed 4 encoder blocks and 4 decoder blocks. Other parameters to tune (kernel, activation, number of convolution layers) can be accessed via the function arguments.

Besides the fundamental blocks, there are some additional ones of note: **Encoder block 5** serves as the transitional layer before decoding, therefore there is no more dimension reduction (maxpooling). **The last convolutional layer** is a 1x1 layer that uses a SoftMax activation to enable the classification of each pixel into each of the 6 labels.

Lastly, we can initiate the model to be used for our dataset by indicating that there are 6 classes and the input image size is 128 x 128 and in RGB (i.e., x 3). The model summary shows that there is indeed a U-shape in the dimensions of the input image change, with particular focus on blocks where the previous layer input is concatenated with the skip layer input, the defining feature of a U-Net.

In [ ]:
```
def build_unet(n_classes, input_size, activation='elu', n_filters=32, n_conv=2, stride=2, ker

    # Input size represent the size of 1 image (the size used for pre-processing)
    input = Input(input_size)

    # Encoder includes multiple blocks with increasing filter.
    eblock1 = EncoderBlock(input, n_conv, activation, n_filters, kernel, stride, max_pooling=
    eblock2 = EncoderBlock(eblock1[0], n_conv, activation, n_filters*2, kernel, stride, max_p
    eblock3 = EncoderBlock(eblock2[0], n_conv, activation, n_filters*4, kernel, stride, max_p
    eblock4 = EncoderBlock(eblock3[0], n_conv, activation, n_filters*8, kernel, stride, max_p

    # Bottleneck layer, last encoder block, no maxpooling
    eblock5 = EncoderBlock(eblock4[0], n_conv, activation, n_filters*16, kernel, stride, max_

    # Decoder includes multiple blocks with decreasing filters
    dblock6 = DecoderBlock(eblock5[0], eblock4[1], n_conv, activation, n_filters*8,kernel,str
    dblock7 = DecoderBlock(dblock6, eblock3[1], n_conv, activation, n_filters*4,kernel,stride
    dblock8 = DecoderBlock(dblock7, eblock2[1], n_conv, activation, n_filters*2,kernel,stride
    dblock9 = DecoderBlock(dblock8, eblock1[1], n_conv, activation, n_filters,kernel,stride)

    # Complete the model with a 1x1 Conv layer (merge channels) and softmax activation for cl
    output = Conv2D(n_classes, kernel_size=1, padding='same',activation='softmax')(dblock9)

    # Define the model
    model = Model(inputs=input, outputs=output)

    return model
```

In [ ]:
```
test_model = build_unet(n_classes=6, input_size=target_img)
test_model.summary()
```

```
Model: "functional_1"
```

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 128, 128, 3)	0	-
conv2d (Conv2D)	(None, 128, 128, 32)	896	input_layer[0][0]
batch_normalization (BatchNormalizatio…	(None, 128, 128, 32)	128	conv2d[0][0]
conv2d_1 (Conv2D)	(None, 128, 128, 32)	9,248	batch_normalizat…
conv2d_2 (Conv2D)	(None, 128, 128, 32)	9,248	conv2d_1[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 128, 128, 32)	128	conv2d_2[0][0]
conv2d_3 (Conv2D)	(None, 128, 128, 32)	9,248	batch_normalizat…
dropout (Dropout)	(None, 128, 128, 32)	0	conv2d_3[0][0]
max_pooling2d (MaxPooling2D)	(None, 64, 64, 32)	0	dropout[0][0]
conv2d_4 (Conv2D)	(None, 64, 64, 64)	18,496	max_pooling2d[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 64, 64, 64)	256	conv2d_4[0][0]
conv2d_5 (Conv2D)	(None, 64, 64, 64)	36,928	batch_normalizat…
conv2d_6 (Conv2D)	(None, 64, 64, 64)	36,928	conv2d_5[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 64, 64, 64)	256	conv2d_6[0][0]
conv2d_7 (Conv2D)	(None, 64, 64, 64)	36,928	batch_normalizat…
dropout_1 (Dropout)	(None, 64, 64, 64)	0	conv2d_7[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 32, 32, 64)	0	dropout_1[0][0]
conv2d_8 (Conv2D)	(None, 32, 32, 128)	73,856	max_pooling2d_1[…
batch_normalizatio… (BatchNormalizatio…	(None, 32, 32, 128)	512	conv2d_8[0][0]
conv2d_9 (Conv2D)	(None, 32, 32,	147,584	batch_normalizat…

	128)		
conv2d_10 (Conv2D)	(None, 32, 32, 128)	147,584	conv2d_9[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 32, 32, 128)	512	conv2d_10[0][0]
conv2d_11 (Conv2D)	(None, 32, 32, 128)	147,584	batch_normalizat…
dropout_2 (Dropout)	(None, 32, 32, 128)	0	conv2d_11[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 16, 16, 128)	0	dropout_2[0][0]
conv2d_12 (Conv2D)	(None, 16, 16, 256)	295,168	max_pooling2d_2[…
batch_normalizatio… (BatchNormalizatio…	(None, 16, 16, 256)	1,024	conv2d_12[0][0]
conv2d_13 (Conv2D)	(None, 16, 16, 256)	590,080	batch_normalizat…
conv2d_14 (Conv2D)	(None, 16, 16, 256)	590,080	conv2d_13[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 16, 16, 256)	1,024	conv2d_14[0][0]
conv2d_15 (Conv2D)	(None, 16, 16, 256)	590,080	batch_normalizat…
dropout_3 (Dropout)	(None, 16, 16, 256)	0	conv2d_15[0][0]
max_pooling2d_3 (MaxPooling2D)	(None, 8, 8, 256)	0	dropout_3[0][0]
conv2d_16 (Conv2D)	(None, 8, 8, 512)	1,180,160	max_pooling2d_3[…
batch_normalizatio… (BatchNormalizatio…	(None, 8, 8, 512)	2,048	conv2d_16[0][0]
conv2d_17 (Conv2D)	(None, 8, 8, 512)	2,359,808	batch_normalizat…
conv2d_18 (Conv2D)	(None, 8, 8, 512)	2,359,808	conv2d_17[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 8, 8, 512)	2,048	conv2d_18[0][0]
conv2d_19 (Conv2D)	(None, 8, 8, 512)	2,359,808	batch_normalizat…
dropout_4 (Dropout)	(None, 8, 8, 512)	0	conv2d_19[0][0]
conv2d_transpose (Conv2DTranspose)	(None, 16, 16, 256)	1,179,904	dropout_4[0][0]
concatenate	(None, 16, 16,	0	conv2d_transpose…

(Concatenate)	512)		conv2d_15[0][0]
conv2d_20 (Conv2D)	(None, 16, 16, 256)	1,179,904	concatenate[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 16, 16, 256)	1,024	conv2d_20[0][0]
conv2d_21 (Conv2D)	(None, 16, 16, 256)	590,080	batch_normalizat…
conv2d_22 (Conv2D)	(None, 16, 16, 256)	590,080	conv2d_21[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 16, 16, 256)	1,024	conv2d_22[0][0]
conv2d_23 (Conv2D)	(None, 16, 16, 256)	590,080	batch_normalizat…
conv2d_transpose_1 (Conv2DTranspose)	(None, 32, 32, 128)	295,040	conv2d_23[0][0]
concatenate_1 (Concatenate)	(None, 32, 32, 256)	0	conv2d_transpose… conv2d_11[0][0]
conv2d_24 (Conv2D)	(None, 32, 32, 128)	295,040	concatenate_1[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 32, 32, 128)	512	conv2d_24[0][0]
conv2d_25 (Conv2D)	(None, 32, 32, 128)	147,584	batch_normalizat…
conv2d_26 (Conv2D)	(None, 32, 32, 128)	147,584	conv2d_25[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 32, 32, 128)	512	conv2d_26[0][0]
conv2d_27 (Conv2D)	(None, 32, 32, 128)	147,584	batch_normalizat…
conv2d_transpose_2 (Conv2DTranspose)	(None, 64, 64, 64)	73,792	conv2d_27[0][0]
concatenate_2 (Concatenate)	(None, 64, 64, 128)	0	conv2d_transpose… conv2d_7[0][0]
conv2d_28 (Conv2D)	(None, 64, 64, 64)	73,792	concatenate_2[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 64, 64, 64)	256	conv2d_28[0][0]
conv2d_29 (Conv2D)	(None, 64, 64, 64)	36,928	batch_normalizat…
conv2d_30 (Conv2D)	(None, 64, 64, 64)	36,928	conv2d_29[0][0]

batch_normalizatio… (BatchNormalizatio…	(None, 64, 64, 64)	256	conv2d_30[0][0]
conv2d_31 (Conv2D)	(None, 64, 64, 64)	36,928	batch_normalizat…
conv2d_transpose_3 (Conv2DTranspose)	(None, 128, 128, 32)	18,464	conv2d_31[0][0]
concatenate_3 (Concatenate)	(None, 128, 128, 64)	0	conv2d_transpose… conv2d_3[0][0]
conv2d_32 (Conv2D)	(None, 128, 128, 32)	18,464	concatenate_3[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 128, 128, 32)	128	conv2d_32[0][0]
conv2d_33 (Conv2D)	(None, 128, 128, 32)	9,248	batch_normalizat…
conv2d_34 (Conv2D)	(None, 128, 128, 32)	9,248	conv2d_33[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 128, 128, 32)	128	conv2d_34[0][0]
conv2d_35 (Conv2D)	(None, 128, 128, 32)	9,248	batch_normalizat…
conv2d_36 (Conv2D)	(None, 128, 128, 6)	198	conv2d_35[0][0]

Total params: 16,497,414 (62.93 MB)

Trainable params: 16,491,526 (62.91 MB)

Non-trainable params: 5,888 (23.00 KB)

5. Tuning and training the model

5.1 A manual approach to Grid Search

Initially, `GridSearchCV` in the `scikit-learn` library was to be used to easily tune the different hyperparameters for our model so that we can proceed to the full-scale training with the best headstart. However, it is incompatible with the U-Net built from scratch. Therefore, we introduced a more brute-force approach to GridSearch below, through which we will monitor the accuracy metric.

This is not an ideal solution but thanks to the limited scope (3 parameters to tune), and the relatively small size of our dataset, this can be an acceptable workaround. To make this approach somewhat more efficient, we also employ Early Stopping provided by Tensorflow/Keras. Lastly, since the manual GridSearch will only be done on 10% of the train dataset and for only 50 epochs, there might be bias in favour of models that manage to learn fast from the limited data subset in the beginning but may plateau after that.

In [ ]:

```
# Set the parameter grid
params = {
    'kernel': [3,5],
    'n_conv': [1,2,3], # Note, 1 indicates a pair of conv layers sandwiching a batch normaliz
    'activation': ['relu', 'elu'],
    }

# Reiterate the model building function with different parameters, and train the model, captu
def tune_unet(X, y, params,epochs=100):

    # Store the parameters for each model
    kernels = []
    n_convs = []
    activations = []

    # Store the accuracy and loss for each model
    accuracies = []
    losses = []

    # Calculate the total number of iterations
    total = 1
    for key in params:
        total *= len(params[key])
    count = 0

    # Iterate over all the parameters

    early_stopping = EarlyStopping(patience=5) # stop training after 5 epochs of no improveme

    for activation in params['activation']:
            for n_conv in params['n_conv']:
                for kernel in params['kernel']:

                    # Initialize, compile and train the model
                    model = build_unet(n_classes=6, activation=activation, n_conv=n_conv, ker
                    model.compile(optimizer=Adam(learning_rate=0.001), loss=SparseCategorical
                    results = model.fit(X, y, epochs=epochs, callbacks=[early_stopping],verbo

                    # Store the parameters for each model
                    kernels.append(kernel)
                    n_convs.append(n_conv)
                    activations.append(activation)

                    # Take the last accuracy and its corresponding loss
                    accuracies.append(results.history['accuracy'][-1])
                    losses.append(results.history['loss'][-1])

                    count += 1
                    print(f'{count}/{total} done')
                    print(f'activation: {activation}, kernel: {kernel}, n_conv: {n_conv}, acc

    # Transform the results into a dataframe for easy analysis
    dict_results= {'activation': activations,
                   'kernel': kernels,
                   'n_conv': n_convs,
                    'accuracy': accuracies,
                    'loss': losses}
    df_results = pd.DataFrame(dict_results)

    return df_results
```

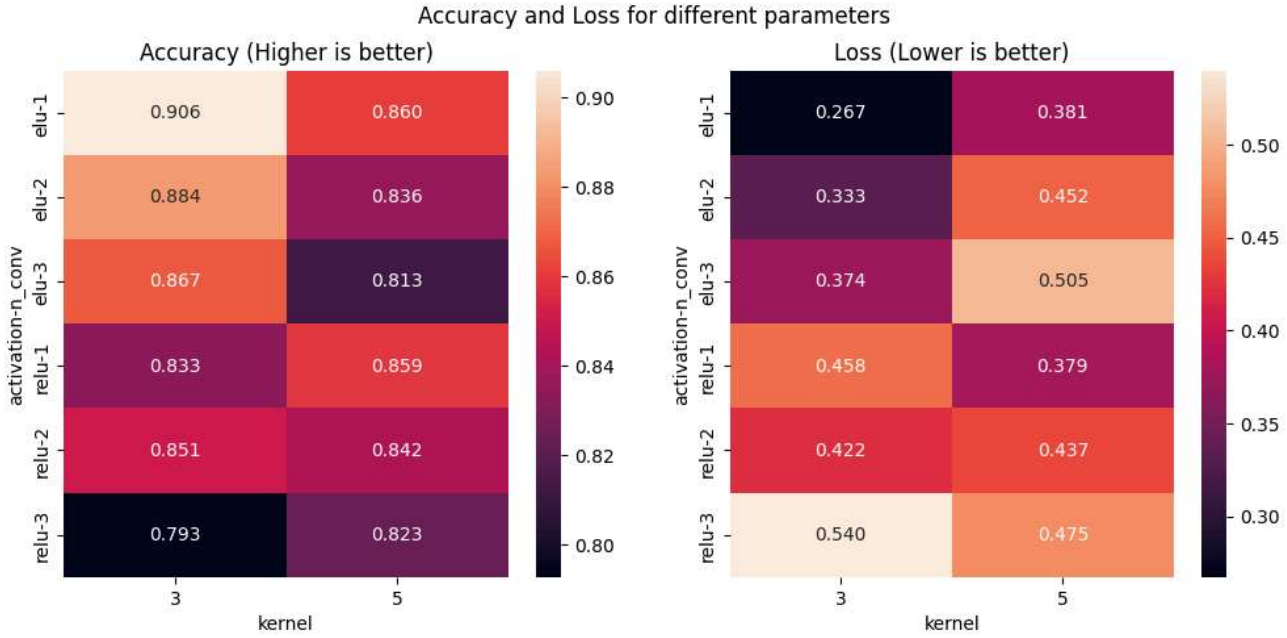
In [ ]:
```
# %%time
# # Randomly select a 10% subset of the train data to speed up the grid search
# X_train_subset, _, y_train_subset, _ = train_test_split(X_train, y_train, test_size=0.90, ra
# # Perform the grid search
# manual_gridsearch = tune_unet(X_train_subset, y_train_subset, params, epochs=50)
```

In [ ]:
```
# # Transform this to a pivot table for better visualization
# pivot_accuracy = manual_gridsearch.pivot_table(index=['activation','n_conv'], columns=['kern
# pivot_loss = manual_gridsearch.pivot_table(index=['activation','n_conv'], columns=['kernel'
```

In [ ]:
```
# # Plot the pivot tables
# fig, ax = plt.subplots(1, 2, figsize=(12, 5))
# sns.heatmap(pivot_accuracy, annot=True, fmt=".3f", ax=ax[0])
# ax[0].set_title('Accuracy (Higher is better)')
# sns.heatmap(pivot_loss, annot=True, fmt=".3f", ax=ax[1])
# ax[1].set_title('Loss (Lower is better)')
# fig.suptitle('Accuracy and Loss for different parameters')
```

Due to the amount of time needed for this manual grid search (120+ mins), here is a summary of accuracy and loss results for different combinations of parameters from the last tuning performed. A key observation here is that more complex architecture (i.e, more conv layer) does not equal better performance.

```
1/12 done
activation: relu, kernel: 3, n_conv: 1, accuracy: 0.8325958251953125, loss: 0.4581790268421173
2/12 done
activation: relu, kernel: 5, n_conv: 1, accuracy: 0.8589380383491516, loss: 0.37926098704338074
3/12 done
activation: relu, kernel: 3, n_conv: 2, accuracy: 0.8506677746772766, loss: 0.4218519926071167
4/12 done
activation: relu, kernel: 5, n_conv: 2, accuracy: 0.8417635560035706, loss: 0.4368758201599121
5/12 done
activation: relu, kernel: 3, n_conv: 3, accuracy: 0.7926469445228577, loss: 0.5398857593536377
6/12 done
activation: relu, kernel: 5, n_conv: 3, accuracy: 0.8234266638755798, loss: 0.4753628671169281
7/12 done
activation: elu, kernel: 3, n_conv: 1, accuracy: 0.9060724377632141, loss: 0.26735973358154297
8/12 done
activation: elu, kernel: 5, n_conv: 1, accuracy: 0.8602863550186157, loss: 0.3810544013977051
9/12 done
activation: elu, kernel: 3, n_conv: 2, accuracy: 0.8837308287620544, loss: 0.332939088344574
10/12 done
activation: elu, kernel: 5, n_conv: 2, accuracy: 0.8357377648353577, loss: 0.4516620337963104
11/12 done
activation: elu, kernel: 3, n_conv: 3, accuracy: 0.8668934106826782, loss: 0.3744097650051117
12/12 done
activation: elu, kernel: 5, n_conv: 3, accuracy: 0.8127940893173218, loss: 0.5053531527519226
CPU times: total: 15h 31min 3s
Wall time: 2h 45min 42s
```



5.2 Training the U-Net with the best parameters

With the results above, we will proceed to training the model in full with the following parameter values:

- Kernel Size: 3
- Pairs of convolution layers per block: 1
- Activation function: ELU

Besides, we will run 200 training epochs with a batch size of 32. Since this is a multi-class segmentation task whereby the labels are integers, we shall use the Sparse Categorical Cross-entropy loss function. Also, the Adam optimiser is used by default. All these are also considered U-Net hyperparameters but their tuning is out of scope for this notebook.

In [ ]:
```
%%time
# Build the model with the best parameters
model = build_unet(n_classes=6, activation='elu', n_conv=1, kernel=3, input_size=target_img)
```

```
model.compile(optimizer=Adam(),
              loss=SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

results = model.fit(X_train, y_train,
                    batch_size=32,
                    epochs=200,
                    validation_data=(X_test, y_test),
                    callbacks=[ModelCheckpoint('best_unet.keras', save_best_only=True)])

fig, (ax_loss, ax_acc) = plt.subplots(1, 2, figsize=(15,5))
ax_loss.plot(results.epoch, results.history["loss"], label="Train loss")
ax_loss.plot(results.epoch, results.history["val_loss"], label="Validation loss")
ax_acc.plot(results.epoch, results.history["accuracy"], label="Train accuracy")
ax_acc.plot(results.epoch, results.history["val_accuracy"], label="Validation accuracy")
ax_loss.legend()
ax_acc.legend()
```

```
Epoch 1/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 30s 2s/step - accuracy: 0.2388 - loss: 2.1076 - val_accuracy: 0.1945
- val_loss: 3.1622
Epoch 2/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 6s 2s/step - accuracy: 0.4841 - loss: 1.3773 - val_accuracy: 0.2098 -
val_loss: 19.5635
Epoch 3/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 6s 2s/step - accuracy: 0.5404 - loss: 1.2221 - val_accuracy: 0.2072 -
val_loss: 5.3467
Epoch 4/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 6s 2s/step - accuracy: 0.5715 - loss: 1.1332 - val_accuracy: 0.2534 -
val_loss: 5.4228
Epoch 5/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.6055 - loss: 1.0646 - val_accuracy: 0.2196 -
val_loss: 8.5227
Epoch 6/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.6226 - loss: 0.9906 - val_accuracy: 0.2698 -
val_loss: 9.0115
Epoch 7/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.6413 - loss: 0.9555 - val_accuracy: 0.2539 -
val_loss: 10.3563
Epoch 8/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.6589 - loss: 0.9247 - val_accuracy: 0.2730 -
val_loss: 12.9663
Epoch 9/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.6701 - loss: 0.8933 - val_accuracy: 0.3553 -
val_loss: 4.0926
Epoch 10/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.6737 - loss: 0.8728 - val_accuracy: 0.2812 -
val_loss: 5.8596
Epoch 11/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.6886 - loss: 0.8271 - val_accuracy: 0.2160 -
val_loss: 9.8120
Epoch 12/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.7009 - loss: 0.7982 - val_accuracy: 0.3408 -
val_loss: 5.1641
Epoch 13/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.7072 - loss: 0.7870 - val_accuracy: 0.3146 -
val_loss: 5.3823
Epoch 14/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.7378 - loss: 0.7274 - val_accuracy: 0.2661 -
val_loss: 4.1422
Epoch 15/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.7243 - loss: 0.7363 - val_accuracy: 0.2252 -
val_loss: 5.7304
Epoch 16/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.7221 - loss: 0.7583 - val_accuracy: 0.3347 -
val_loss: 4.9992
Epoch 17/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.7489 - loss: 0.6928 - val_accuracy: 0.2574 -
val_loss: 11.5090
Epoch 18/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.7540 - loss: 0.6857 - val_accuracy: 0.2382 -
val_loss: 8.9696
Epoch 19/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.7650 - loss: 0.6478 - val_accuracy: 0.2000 -
val_loss: 15.9428
Epoch 20/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.7585 - loss: 0.6521 - val_accuracy: 0.2416 -
val_loss: 5.8837
Epoch 21/200
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.7692 - loss: 0.6219 - val_accuracy: 0.2084 -
val_loss: 10.4405
Epoch 22/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.7866 - loss: 0.5865 - val_accuracy: 0.2256 -
val_loss: 8.7409
Epoch 23/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.7884 - loss: 0.5729 - val_accuracy: 0.2415 -
val_loss: 7.2735
Epoch 24/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.7906 - loss: 0.5663 - val_accuracy: 0.2298 -
val_loss: 6.2925
Epoch 25/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8033 - loss: 0.5391 - val_accuracy: 0.2154 -
val_loss: 8.9469
Epoch 26/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8075 - loss: 0.5278 - val_accuracy: 0.2110 -
val_loss: 9.2337
Epoch 27/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8067 - loss: 0.5197 - val_accuracy: 0.2662 -
val_loss: 4.8889
Epoch 28/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8054 - loss: 0.5325 - val_accuracy: 0.2427 -
val_loss: 5.0531
Epoch 29/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8105 - loss: 0.5184 - val_accuracy: 0.2623 -
val_loss: 6.4421
Epoch 30/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8070 - loss: 0.5240 - val_accuracy: 0.2385 -
val_loss: 10.8835
Epoch 31/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8166 - loss: 0.5029 - val_accuracy: 0.2226 -
val_loss: 9.3248
Epoch 32/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8299 - loss: 0.4669 - val_accuracy: 0.2500 -
val_loss: 6.2981
Epoch 33/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8339 - loss: 0.4543 - val_accuracy: 0.2378 -
val_loss: 6.0686
Epoch 34/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8366 - loss: 0.4426 - val_accuracy: 0.2669 -
val_loss: 4.0176
Epoch 35/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8420 - loss: 0.4270 - val_accuracy: 0.2747 -
val_loss: 3.5187
Epoch 36/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8469 - loss: 0.4204 - val_accuracy: 0.2267 -
val_loss: 4.6028
Epoch 37/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8539 - loss: 0.4010 - val_accuracy: 0.2427 -
val_loss: 4.6421
Epoch 38/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8556 - loss: 0.3925 - val_accuracy: 0.2339 -
val_loss: 4.5691
Epoch 39/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.8588 - loss: 0.3839 - val_accuracy: 0.2300 -
val_loss: 4.6390
Epoch 40/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.8573 - loss: 0.3865 - val_accuracy: 0.2282 -
val_loss: 4.5387
Epoch 41/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.8675 - loss: 0.3676 - val_accuracy: 0.3635 -
```

```
val_loss: 3.0922
Epoch 42/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8631 - loss: 0.3723 - val_accuracy: 0.2508 -
val_loss: 4.3715
Epoch 43/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8636 - loss: 0.3714 - val_accuracy: 0.3001 -
val_loss: 3.8192
Epoch 44/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8664 - loss: 0.3633 - val_accuracy: 0.3024 -
val_loss: 3.5059
Epoch 45/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8686 - loss: 0.3557 - val_accuracy: 0.2770 -
val_loss: 3.6725
Epoch 46/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8682 - loss: 0.3616 - val_accuracy: 0.2613 -
val_loss: 4.0978
Epoch 47/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8780 - loss: 0.3339 - val_accuracy: 0.2932 -
val_loss: 3.4552
Epoch 48/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8794 - loss: 0.3296 - val_accuracy: 0.2811 -
val_loss: 3.4060
Epoch 49/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8805 - loss: 0.3271 - val_accuracy: 0.3041 -
val_loss: 3.2750
Epoch 50/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8838 - loss: 0.3171 - val_accuracy: 0.3222 -
val_loss: 2.8471
Epoch 51/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8865 - loss: 0.3088 - val_accuracy: 0.3287 -
val_loss: 2.6568
Epoch 52/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8840 - loss: 0.3131 - val_accuracy: 0.3380 -
val_loss: 2.7703
Epoch 53/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8868 - loss: 0.3117 - val_accuracy: 0.2965 -
val_loss: 3.1325
Epoch 54/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8884 - loss: 0.3156 - val_accuracy: 0.2745 -
val_loss: 3.9240
Epoch 55/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8736 - loss: 0.3525 - val_accuracy: 0.3072 -
val_loss: 3.6391
Epoch 56/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8711 - loss: 0.3536 - val_accuracy: 0.2996 -
val_loss: 3.7587
Epoch 57/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8680 - loss: 0.3582 - val_accuracy: 0.2700 -
val_loss: 4.7213
Epoch 58/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8798 - loss: 0.3310 - val_accuracy: 0.2570 -
val_loss: 4.9922
Epoch 59/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8832 - loss: 0.3200 - val_accuracy: 0.2519 -
val_loss: 4.6207
Epoch 60/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8886 - loss: 0.3071 - val_accuracy: 0.2611 -
val_loss: 4.2563
Epoch 61/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8929 - loss: 0.2942 - val_accuracy: 0.2570 -
val_loss: 4.0980
```

```
Epoch 62/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8969 - loss: 0.2819 - val_accuracy: 0.2995 -
val_loss: 3.7446
Epoch 63/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9009 - loss: 0.2707 - val_accuracy: 0.3344 -
val_loss: 3.0951
Epoch 64/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8995 - loss: 0.2751 - val_accuracy: 0.3151 -
val_loss: 3.5266
Epoch 65/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.8983 - loss: 0.2764 - val_accuracy: 0.2789 -
val_loss: 3.7778
Epoch 66/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9058 - loss: 0.2552 - val_accuracy: 0.3467 -
val_loss: 3.0853
Epoch 67/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9078 - loss: 0.2506 - val_accuracy: 0.3249 -
val_loss: 3.0468
Epoch 68/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9068 - loss: 0.2562 - val_accuracy: 0.3416 -
val_loss: 3.0433
Epoch 69/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9036 - loss: 0.2589 - val_accuracy: 0.3899 -
val_loss: 2.5313
Epoch 70/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9109 - loss: 0.2439 - val_accuracy: 0.3505 -
val_loss: 2.7959
Epoch 71/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9102 - loss: 0.2422 - val_accuracy: 0.3477 -
val_loss: 3.0549
Epoch 72/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9127 - loss: 0.2368 - val_accuracy: 0.3749 -
val_loss: 2.7683
Epoch 73/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9158 - loss: 0.2290 - val_accuracy: 0.3710 -
val_loss: 2.7990
Epoch 74/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9168 - loss: 0.2254 - val_accuracy: 0.3654 -
val_loss: 2.8972
Epoch 75/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9161 - loss: 0.2300 - val_accuracy: 0.3733 -
val_loss: 2.7153
Epoch 76/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9185 - loss: 0.2203 - val_accuracy: 0.4295 -
val_loss: 2.5159
Epoch 77/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9204 - loss: 0.2145 - val_accuracy: 0.4136 -
val_loss: 2.5606
Epoch 78/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9187 - loss: 0.2193 - val_accuracy: 0.4320 -
val_loss: 2.3543
Epoch 79/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9216 - loss: 0.2126 - val_accuracy: 0.4602 -
val_loss: 2.2841
Epoch 80/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9223 - loss: 0.2116 - val_accuracy: 0.4184 -
val_loss: 2.4734
Epoch 81/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9208 - loss: 0.2123 - val_accuracy: 0.4404 -
val_loss: 2.3606
Epoch 82/200
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9231 - loss: 0.2068 - val_accuracy: 0.4130 -
val_loss: 2.4471
Epoch 83/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9245 - loss: 0.2032 - val_accuracy: 0.4155 -
val_loss: 2.4994
Epoch 84/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9223 - loss: 0.2078 - val_accuracy: 0.4748 -
val_loss: 2.0917
Epoch 85/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 6s 2s/step - accuracy: 0.9274 - loss: 0.1960 - val_accuracy: 0.4481 -
val_loss: 2.1531
Epoch 86/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9282 - loss: 0.1918 - val_accuracy: 0.5148 -
val_loss: 1.9346
Epoch 87/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9254 - loss: 0.1997 - val_accuracy: 0.4697 -
val_loss: 2.1653
Epoch 88/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9295 - loss: 0.1894 - val_accuracy: 0.4791 -
val_loss: 2.1919
Epoch 89/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9309 - loss: 0.1851 - val_accuracy: 0.4739 -
val_loss: 2.2593
Epoch 90/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9316 - loss: 0.1847 - val_accuracy: 0.4992 -
val_loss: 2.0826
Epoch 91/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9319 - loss: 0.1830 - val_accuracy: 0.4861 -
val_loss: 2.1383
Epoch 92/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9315 - loss: 0.1842 - val_accuracy: 0.5089 -
val_loss: 2.0150
Epoch 93/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9350 - loss: 0.1744 - val_accuracy: 0.4983 -
val_loss: 1.9840
Epoch 94/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9349 - loss: 0.1756 - val_accuracy: 0.5026 -
val_loss: 2.1296
Epoch 95/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9310 - loss: 0.1834 - val_accuracy: 0.4904 -
val_loss: 2.2122
Epoch 96/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9336 - loss: 0.1759 - val_accuracy: 0.5148 -
val_loss: 2.0189
Epoch 97/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9337 - loss: 0.1765 - val_accuracy: 0.5168 -
val_loss: 2.0679
Epoch 98/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9343 - loss: 0.1761 - val_accuracy: 0.5159 -
val_loss: 2.0337
Epoch 99/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9365 - loss: 0.1697 - val_accuracy: 0.5451 -
val_loss: 1.8115
Epoch 100/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9390 - loss: 0.1626 - val_accuracy: 0.5406 -
val_loss: 1.8673
Epoch 101/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9414 - loss: 0.1580 - val_accuracy: 0.5534 -
val_loss: 1.8613
Epoch 102/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9376 - loss: 0.1647 - val_accuracy: 0.5463 -
```

```
val_loss: 1.8914
Epoch 103/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9376 - loss: 0.1658 - val_accuracy: 0.5433 -
val_loss: 1.9056
Epoch 104/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9417 - loss: 0.1569 - val_accuracy: 0.5653 -
val_loss: 1.7532
Epoch 105/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9414 - loss: 0.1556 - val_accuracy: 0.5417 -
val_loss: 1.8796
Epoch 106/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9403 - loss: 0.1580 - val_accuracy: 0.5762 -
val_loss: 1.6932
Epoch 107/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9410 - loss: 0.1570 - val_accuracy: 0.5690 -
val_loss: 1.7639
Epoch 108/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9401 - loss: 0.1611 - val_accuracy: 0.5538 -
val_loss: 1.8088
Epoch 109/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9369 - loss: 0.1663 - val_accuracy: 0.5875 -
val_loss: 1.6110
Epoch 110/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9377 - loss: 0.1664 - val_accuracy: 0.5699 -
val_loss: 1.7037
Epoch 111/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9373 - loss: 0.1668 - val_accuracy: 0.5733 -
val_loss: 1.7658
Epoch 112/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9401 - loss: 0.1582 - val_accuracy: 0.5638 -
val_loss: 1.7729
Epoch 113/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9427 - loss: 0.1534 - val_accuracy: 0.5953 -
val_loss: 1.6270
Epoch 114/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9412 - loss: 0.1569 - val_accuracy: 0.5854 -
val_loss: 1.6117
Epoch 115/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9449 - loss: 0.1479 - val_accuracy: 0.6094 -
val_loss: 1.5143
Epoch 116/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9433 - loss: 0.1501 - val_accuracy: 0.5999 -
val_loss: 1.6081
Epoch 117/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9462 - loss: 0.1441 - val_accuracy: 0.6014 -
val_loss: 1.5968
Epoch 118/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9443 - loss: 0.1476 - val_accuracy: 0.6197 -
val_loss: 1.5194
Epoch 119/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9403 - loss: 0.1564 - val_accuracy: 0.6187 -
val_loss: 1.4908
Epoch 120/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9450 - loss: 0.1470 - val_accuracy: 0.6083 -
val_loss: 1.5369
Epoch 121/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9465 - loss: 0.1423 - val_accuracy: 0.6370 -
val_loss: 1.3662
Epoch 122/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9474 - loss: 0.1400 - val_accuracy: 0.5792 -
val_loss: 1.7651
```

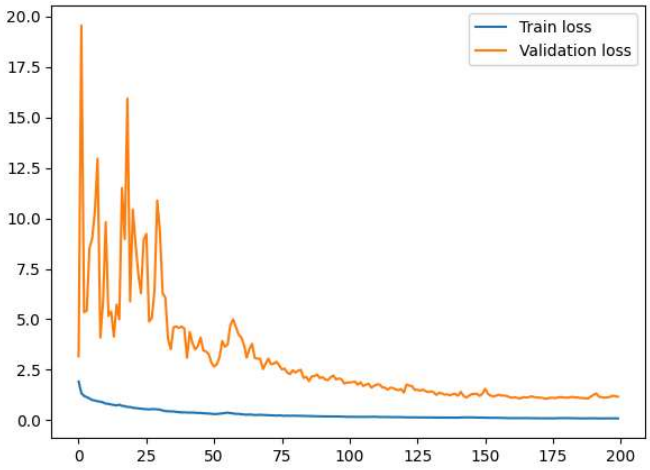
```
Epoch 123/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9474 - loss: 0.1400 - val_accuracy: 0.5950 -
val_loss: 1.7084
Epoch 124/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9460 - loss: 0.1423 - val_accuracy: 0.6002 -
val_loss: 1.6912
Epoch 125/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9476 - loss: 0.1380 - val_accuracy: 0.6270 -
val_loss: 1.4983
Epoch 126/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9496 - loss: 0.1348 - val_accuracy: 0.6169 -
val_loss: 1.5059
Epoch 127/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9502 - loss: 0.1325 - val_accuracy: 0.6357 -
val_loss: 1.4618
Epoch 128/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9477 - loss: 0.1372 - val_accuracy: 0.6172 -
val_loss: 1.5147
Epoch 129/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9497 - loss: 0.1332 - val_accuracy: 0.6365 -
val_loss: 1.4374
Epoch 130/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9490 - loss: 0.1347 - val_accuracy: 0.6384 -
val_loss: 1.3936
Epoch 131/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9487 - loss: 0.1351 - val_accuracy: 0.6350 -
val_loss: 1.4268
Epoch 132/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9518 - loss: 0.1281 - val_accuracy: 0.6414 -
val_loss: 1.3819
Epoch 133/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9515 - loss: 0.1280 - val_accuracy: 0.6631 -
val_loss: 1.2526
Epoch 134/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9491 - loss: 0.1345 - val_accuracy: 0.6460 -
val_loss: 1.3586
Epoch 135/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9487 - loss: 0.1342 - val_accuracy: 0.6554 -
val_loss: 1.3271
Epoch 136/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9529 - loss: 0.1239 - val_accuracy: 0.6660 -
val_loss: 1.2692
Epoch 137/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9520 - loss: 0.1261 - val_accuracy: 0.6610 -
val_loss: 1.2798
Epoch 138/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9538 - loss: 0.1233 - val_accuracy: 0.6713 -
val_loss: 1.2285
Epoch 139/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9548 - loss: 0.1212 - val_accuracy: 0.6661 -
val_loss: 1.2890
Epoch 140/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9515 - loss: 0.1288 - val_accuracy: 0.6573 -
val_loss: 1.2983
Epoch 141/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9547 - loss: 0.1208 - val_accuracy: 0.6831 -
val_loss: 1.2104
Epoch 142/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9482 - loss: 0.1375 - val_accuracy: 0.6498 -
val_loss: 1.4030
Epoch 143/200
```

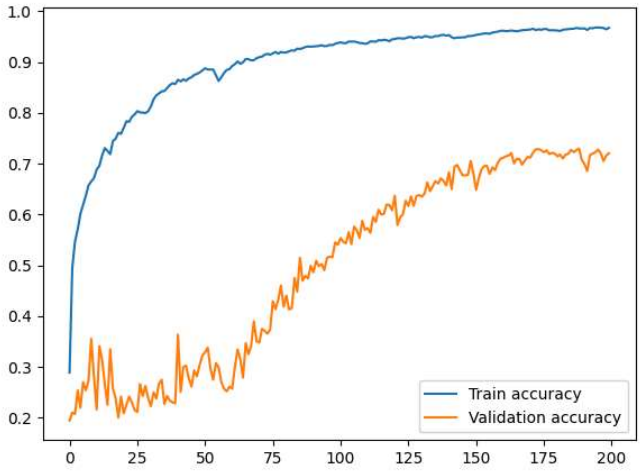
```
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9473 - loss: 0.1388 - val_accuracy: 0.6940 -
val_loss: 1.2035
Epoch 144/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9510 - loss: 0.1307 - val_accuracy: 0.6975 -
val_loss: 1.1201
Epoch 145/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9497 - loss: 0.1338 - val_accuracy: 0.6876 -
val_loss: 1.2078
Epoch 146/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9489 - loss: 0.1356 - val_accuracy: 0.6771 -
val_loss: 1.2894
Epoch 147/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9488 - loss: 0.1381 - val_accuracy: 0.6773 -
val_loss: 1.3040
Epoch 148/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9512 - loss: 0.1274 - val_accuracy: 0.6780 -
val_loss: 1.3042
Epoch 149/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9524 - loss: 0.1269 - val_accuracy: 0.7054 -
val_loss: 1.1960
Epoch 150/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9536 - loss: 0.1221 - val_accuracy: 0.6809 -
val_loss: 1.3205
Epoch 151/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9543 - loss: 0.1203 - val_accuracy: 0.6486 -
val_loss: 1.5503
Epoch 152/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9541 - loss: 0.1202 - val_accuracy: 0.6712 -
val_loss: 1.3177
Epoch 153/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9558 - loss: 0.1158 - val_accuracy: 0.6890 -
val_loss: 1.2111
Epoch 154/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9581 - loss: 0.1108 - val_accuracy: 0.6953 -
val_loss: 1.1801
Epoch 155/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9589 - loss: 0.1104 - val_accuracy: 0.6959 -
val_loss: 1.2104
Epoch 156/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9546 - loss: 0.1186 - val_accuracy: 0.6800 -
val_loss: 1.2635
Epoch 157/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9571 - loss: 0.1119 - val_accuracy: 0.6927 -
val_loss: 1.2172
Epoch 158/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9586 - loss: 0.1087 - val_accuracy: 0.6874 -
val_loss: 1.2255
Epoch 159/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9599 - loss: 0.1058 - val_accuracy: 0.7017 -
val_loss: 1.1969
Epoch 160/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9618 - loss: 0.1012 - val_accuracy: 0.7100 -
val_loss: 1.1376
Epoch 161/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9624 - loss: 0.1002 - val_accuracy: 0.7117 -
val_loss: 1.1145
Epoch 162/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9618 - loss: 0.1023 - val_accuracy: 0.7147 -
val_loss: 1.1325
Epoch 163/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9615 - loss: 0.1014 - val_accuracy: 0.7161 -
```

```
val_loss: 1.0970
Epoch 164/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9631 - loss: 0.0982 - val_accuracy: 0.7213 -
val_loss: 1.0758
Epoch 165/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9640 - loss: 0.0966 - val_accuracy: 0.7007 -
val_loss: 1.1436
Epoch 166/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9611 - loss: 0.1020 - val_accuracy: 0.7098 -
val_loss: 1.1227
Epoch 167/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9631 - loss: 0.0984 - val_accuracy: 0.7090 -
val_loss: 1.1363
Epoch 168/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9625 - loss: 0.0991 - val_accuracy: 0.6982 -
val_loss: 1.1842
Epoch 169/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9637 - loss: 0.0964 - val_accuracy: 0.7062 -
val_loss: 1.1460
Epoch 170/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9633 - loss: 0.0965 - val_accuracy: 0.7136 -
val_loss: 1.1251
Epoch 171/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9651 - loss: 0.0931 - val_accuracy: 0.7119 -
val_loss: 1.1192
Epoch 172/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9660 - loss: 0.0902 - val_accuracy: 0.7216 -
val_loss: 1.1059
Epoch 173/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9627 - loss: 0.0976 - val_accuracy: 0.7283 -
val_loss: 1.0669
Epoch 174/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9647 - loss: 0.0934 - val_accuracy: 0.7288 -
val_loss: 1.0749
Epoch 175/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9632 - loss: 0.0966 - val_accuracy: 0.7269 -
val_loss: 1.1125
Epoch 176/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9647 - loss: 0.0936 - val_accuracy: 0.7225 -
val_loss: 1.1085
Epoch 177/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9656 - loss: 0.0905 - val_accuracy: 0.7264 -
val_loss: 1.0988
Epoch 178/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9640 - loss: 0.0966 - val_accuracy: 0.7192 -
val_loss: 1.1305
Epoch 179/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9634 - loss: 0.0964 - val_accuracy: 0.7213 -
val_loss: 1.1412
Epoch 180/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9645 - loss: 0.0956 - val_accuracy: 0.7203 -
val_loss: 1.1224
Epoch 181/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9632 - loss: 0.0977 - val_accuracy: 0.7145 -
val_loss: 1.1172
Epoch 182/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9633 - loss: 0.0969 - val_accuracy: 0.7181 -
val_loss: 1.1207
Epoch 183/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9643 - loss: 0.0944 - val_accuracy: 0.7104 -
val_loss: 1.1585
```

```
Epoch 184/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9645 - loss: 0.0927 - val_accuracy: 0.7180 -
val_loss: 1.1228
Epoch 185/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9658 - loss: 0.0914 - val_accuracy: 0.7195 -
val_loss: 1.1304
Epoch 186/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9650 - loss: 0.0916 - val_accuracy: 0.7275 -
val_loss: 1.1023
Epoch 187/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9663 - loss: 0.0889 - val_accuracy: 0.7227 -
val_loss: 1.0920
Epoch 188/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9679 - loss: 0.0859 - val_accuracy: 0.7267 -
val_loss: 1.0866
Epoch 189/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9682 - loss: 0.0860 - val_accuracy: 0.7297 -
val_loss: 1.0764
Epoch 190/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9677 - loss: 0.0865 - val_accuracy: 0.7073 -
val_loss: 1.1742
Epoch 191/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9661 - loss: 0.0897 - val_accuracy: 0.6994 -
val_loss: 1.2547
Epoch 192/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step - accuracy: 0.9635 - loss: 0.0962 - val_accuracy: 0.6856 -
val_loss: 1.3231
Epoch 193/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9680 - loss: 0.0847 - val_accuracy: 0.7169 -
val_loss: 1.1590
Epoch 194/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9671 - loss: 0.0864 - val_accuracy: 0.7204 -
val_loss: 1.1395
Epoch 195/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9679 - loss: 0.0847 - val_accuracy: 0.7223 -
val_loss: 1.1194
Epoch 196/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9693 - loss: 0.0814 - val_accuracy: 0.7279 -
val_loss: 1.1251
Epoch 197/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9689 - loss: 0.0818 - val_accuracy: 0.7207 -
val_loss: 1.1505
Epoch 198/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9676 - loss: 0.0847 - val_accuracy: 0.7052 -
val_loss: 1.2096
Epoch 199/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9662 - loss: 0.0884 - val_accuracy: 0.7159 -
val_loss: 1.1936
Epoch 200/200
4/4 ━━━━━━━━━━━━━━━━━━━━ 8s 2s/step - accuracy: 0.9676 - loss: 0.0847 - val_accuracy: 0.7206 -
val_loss: 1.1637
CPU times: total: 2h 4min 40s
Wall time: 24min 3s
```

Out[ ]: <matplotlib.legend.Legend at 0x26323353f80>





5.3 Evaluating the model's performance

Although the model learnt quickly from the train dataset, when validated with the test dataset, it plateaued at approximately **70% accuracy** on the test data after 150 epochs.

The discrepancy between training and testing indicates that there may be overfitting creating bias. Possible reasons for high bias might be due to the limited data at hand, and also to the fact that the model is learning from scratch without relying on any pre-trained backbone. Another reason for high bias and variance may be due to the large number of trainable parameters in the complex architecture of a U-Net.

In [ ]:
```
# Create a table of OA, UA, PA, and F1 scores for each class

def Metrics():
    y_pred = model.predict(X_test)
    y_pred_argmax = np.argmax(y_pred, axis=3)
    y_test_argmax = y_test[:,:,:,0]

    # Flatten the arrays
    y_pred_argmax = y_pred_argmax.flatten()
    y_test_argmax = y_test_argmax.flatten()

    # Calculate the confusion matrix
    cm = confusion_matrix(y_test_argmax, y_pred_argmax)

    # Calculate the overall accuracy
    oa = np.sum(np.diag(cm))/np.sum(cm)

    # Calculate the user's accuracy
    ua = np.diag(cm)/np.sum(cm, axis=0)

    # Calculate the producer's accuracy
    pa = np.diag(cm)/np.sum(cm, axis=1)

    # Calculate the F1 score
    f1 = 2*pa*ua/(pa+ua)

    # Create a table of the metrics
    df_metrics = pd.DataFrame({'Overall Accuracy': oa, 'User Accuracy': ua, 'Producer Accuracy

    return df_metrics
```

```
Metrics()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step
```

Out[ ]:

	Overall Accuracy	User Accuracy	Producer Accuracy	F1 Score
other	0.720583	0.541468	0.604309	0.571165
facade	0.720583	0.773626	0.459378	0.576457
road	0.720583	0.726436	0.676387	0.700519
vegetation	0.720583	0.827307	0.888033	0.856595
vehicle	0.720583	0.753743	0.206479	0.324159
roof	0.720583	0.725038	0.825709	0.772106

The accuracy metric table below further provides insights on the prediction power of the model we have trained:

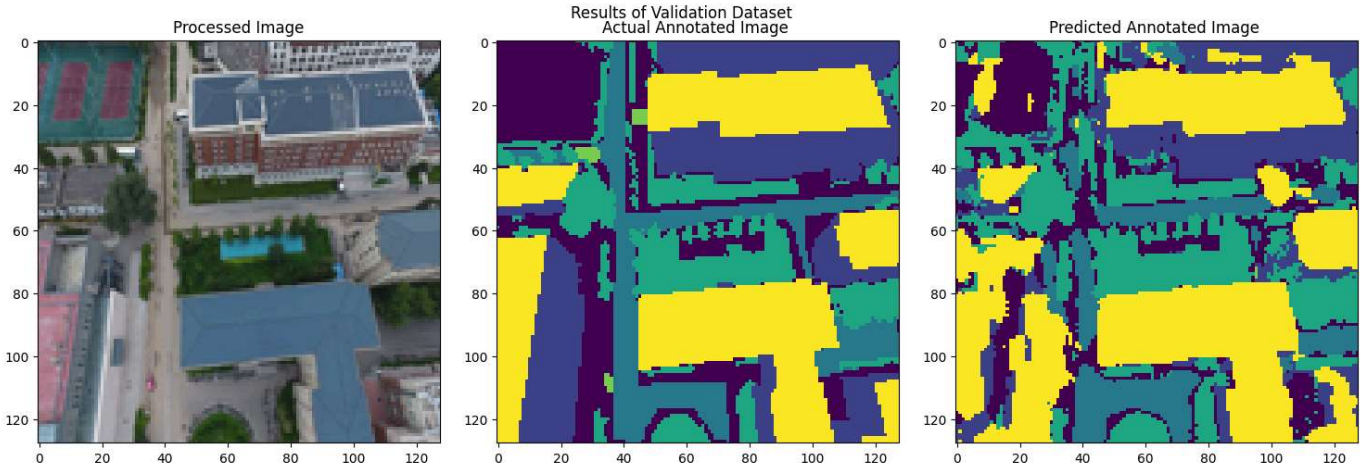
- **Class 0 (other)** has low User Accuracy (high false positives), possibly due to the large variances in the pixel RGB values of miscellaneous objects leading to many pixels being incorrectly predicted as 'other'
- **Class 1 (facade) and 4 (vehicle)** have low Producer Accuracy (high false negatives), possibly due to lack of data leading to incorrect labeling of other facades and vehicles unseen.
- **Class 2 (road), 3 (vegetation), and 5 (roof)** have UA and PA more in line with Overall Accuracy, possibly thanks to large enough train data for these values. For Class 2 (Road), the rather uniform RGB values of the source image made learning them much easier.

Finally, we shall visually inspect the prediction in contrast with the ground truth to deduce any obvious patterns or discrepancies. We can confirm with the confusion matrix that Class 3 (vegetation) and 5 (roof) have more false positives owing to its relative larger class weight in the dataset over others. One possible fix is to initiate a distribution of class weights that is closer to the 'real' distribution, instead of using the 'he_normalised' initaliser.

In [ ]:

```
# Results of Validation Dataset
def VisualizeResults(index):
    img = X_test[index]
    img = img[np.newaxis, ...]
    pred_y = model.predict(img)
    pred_ann = tf.argmax(pred_y[0], axis=-1)
    pred_ann = pred_ann[..., tf.newaxis]
    fig, arr = plt.subplots(1, 3, figsize=(15, 5))
    arr[0].imshow(X_test[index])
    arr[0].set_title('Processed Image')
    arr[1].imshow(y_test[index,:,:,0])
    arr[1].set_title('Actual Annotated Image ')
    arr[2].imshow(pred_ann[:,:,0])
    arr[2].set_title('Predicted Annotated Image ')
    fig.suptitle('Results of Validation Dataset')
    fig.tight_layout()

# Visualise random results
index = np.random.randint(0, len(X_test))
VisualizeResults(index)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 123ms/step
```



6. Discussion

In this notebook, we have constructed a U-Net from its building blocks using Tensorflow/Keras, based on Ronneberger et al., 2015. We then used to perform semantic segmentation tasks on the Urban Drone Dataset with aerial imagery. Furthermore, we configured a brute-force parameter tuning function to choose the best parameters to initiate the models with.

Despite acquiring high training accuracy, the discrepancy between training and validation shows signs of overfitting, requiring a reassessment of whether U-Net is the appropriate architecture for this task. In future iterations, improvements could be made in the following ways:

- For high resolution image dataset like this, instead of resizing (and thus also warping) the inputs, it may be better to crop them into smaller pieces so that we are not downsampling even before training the model. However, when cropping, one must also take care of patching the predictions together into the original dimension.
- Many of the images in the dataset are almost identical because these drone photos might have been taken too close together, which limits the variations needed to train the model. Therefore, image augmentation during pre-processing may be necessary to enrich the dataset further.
- This U-Net can benefit from a more resource-efficient GridSearch, to which we could introduce more hyperparameters to tune, such as optimisers, loss functions, batch size, epochs, etc., similar to what has been done by Tran Thanh (2021)
- Besides U-Net, SegNet is also a highly recommended architecture for semantic segmentation tasks (Wang et al. 2023). Therefore, a comparison between the two architectures side-by-side may offer further insights into what might work best for this aerial imagery dataset.

References

- Chen, Yu & Wang, Yao & Lu, Peng & Yisong, Chen & Wang, Guoping. (2018). Large-Scale Structure from Motion with Semantic Constraints of Aerial Images: First Chinese Conference, PRCV 2018, Guangzhou, China, November 23-26, 2018, Proceedings, Part I. 347-359. 10.1007/978-3-030-03398-9_30. https://doi.org/10.1007/978-3-030-03398-9_30
- Lee, Yongkyu, Woodam Sim, Jeongmook Park, and Jungsoo Lee. 2022. "Evaluation of Hyperparameter Combinations of the U-Net Model for Land Cover Classification" Forests 13, no. 11: 1813. https://doi.org/10.3390/f13111813

- Olaf Ronneberger and Philipp Fischer and Thomas Brox. (2015) U-Net: Convolutional Networks for Biomedical Image Segmentation. 1505.04597. https://doi.org/10.1016/j.compeleceng.2023.108734
- Thanh, Tran. (2021). Grid Search of Convolutional Neural Network model in the case of load forecasting. Archives of Electrical Engineering. 70. 10.24425/aee.2021.136050. https://doi.org/10.24425/aee.2021.136050
- Xin Wang, Shihan Jing, Huifeng Dai, Aiye Shi. (2023). High-resolution remote sensing images semantic segmentation using improved UNet and SegNet,Computers and Electrical Engineering, Volume 108, 2023, 108734, ISSN 0045-7906. 10.1016/j.compeleceng.2023.108734. https://doi.org/10.1016/j.compeleceng.2023.108734
- Use of GitHub Copilot AI functionality.