DETECTING CROP ROWS FROM IMAGE DATA

Introduction:

Deep learning-based approach can be used for object detection and segmentation in image data. Deep learning algorithms can identify patterns in images even with variations of light levels and minor occlusions. They can therefore detect crop rows despite the variations in light levels, size and placement of the objects, and variations in the background of desired subject (crop rows). We can detect high level semantic features such as crop rows rather than just isolating the pixels belonging to the crop. The deep learning-based crop row detection approach involves using image segmentation to predict a binary mask for the crop rows, which can be used to detect the parameters defining lines corresponding to crop rows.

Machine learning based methods for crop row detection:

Machine learning based methods often use deep neural networks to learn a latent representation of the crop rows. Here there are two types: supervised and unsupervised learning. In supervised learning, the model is trained on labeled data where the crop rows are annotated with ground truth labels. Unsupervised learning methods try to learn a representation of the crop rows without any ground truth labels. However, unsupervised methods may not perform as well as supervised methods, especially if the crop rows have complex shapes and structures.

A model is trained to learn the features that are most important for identifying crop rows in images. CNNs are the most popular type of neural network used for image analysis and therefore can be used for crop row detection. These models can automatically learn a hierarchical set of features that are optimal for detecting crop rows in images. U-Net is one of the most popular CNN-based model for crop row detection which is an encoder-decoder network that uses skip connections to preserve spatial information.

Dataset:

The image dataset consists of both training and test images. The images are 320x240 JPEG images with 3 channels (RGB). These images are then converted to numpy arrays and flattened to 1D arrays that contain pixel information. The labels for the training data are stored in the "train_labels" folder. The test labels are also stored as images, but they are grayscale (one channel) and only contain two pixel values - 0 and 255.

Total images=281

width = 320, height = 240, and channels = 3 (for RGB images).

Random augmentations is a common technique to increase the size and variability of a small dataset. This can help prevent overfitting, as it encourages the network to learn more general features that are robust to variations in the input data. Augmentations such as rotations, flips, and brightness altering can help expose the network to different aspects of the data, which can improve its ability to generalize to new examples. By prioritizing a larger portion of the dataset for training, we can also ensure that the network has sufficient data to learn the underlying patterns of the crop rows.

Methodology:

I am using supervised learning method for crop row detection. Specifically, given an input image, the approach classifies regions of pixels into either of two classes: crop-row or background. This is known as semantic segmentation. To perform this task, the approach used is a U-Net based neural network architecture.

The U-Net architecture is designed to be relatively lightweight in terms of the number of layers and trainable parameters. This is done to prevent overfitting to the small training set that is available. After performing semantic segmentation, the approach uses density-based clustering of crop-row pixels to detect clusters that resemble crop rows. These clusters are identified in the binary segmented image space where crop-row pixels are designated as pixels with a value of 1.

It involves training a neural network to learn the mapping between input images and their corresponding ground truth images, which accurately locate and represent the geometric shape of crop rows within the region of interest.

Overall, this approach uses neural network-based semantic segmentation.

U-Net architecture:

This architecture adds skip connections between the encoder and decoder layers, allowing the decoder to access information from earlier stages of the encoding process. This helps to improve the output segmentation map with finer features. The architecture and composition of layers within the encoder and decoder blocks is shown in Figure 1.

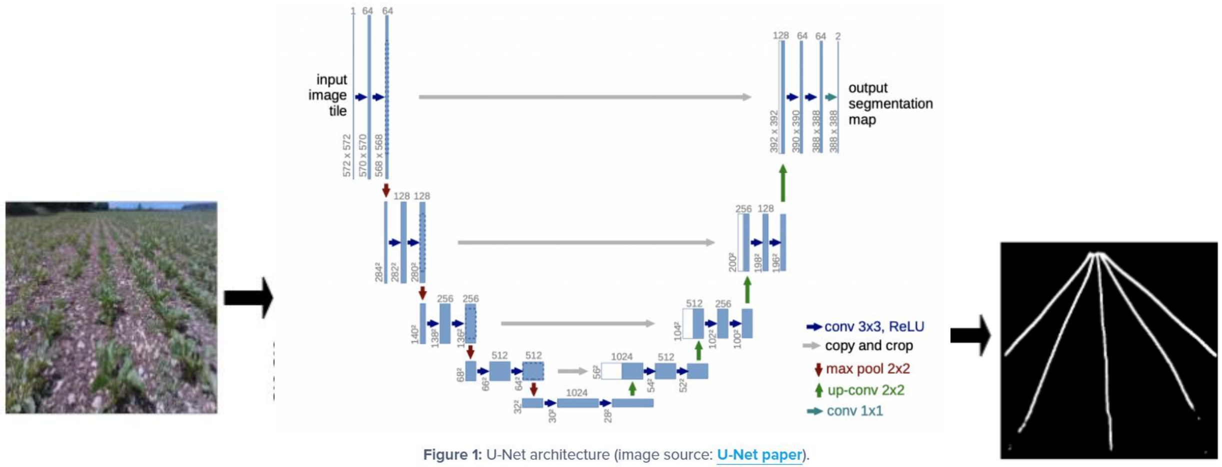


Figure 1: U-Net architecture (image source: U-Net paper).

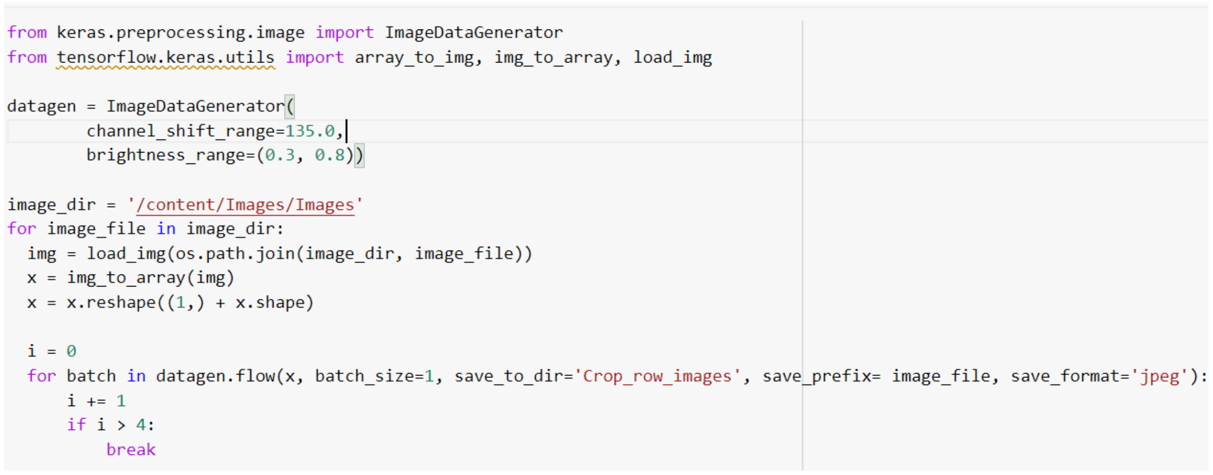
The model architecture is simple: an encoder (for down sampling) and a decoder (for up sampling) with skip connections. As Figure 1 shows, it is shaped like the letter U hence the name U-Net. The gray arrows indicate the skip connections that concatenate the encoder feature map with the decoder, which helps the backward flow of gradients for improved training.

The architecture shows that an input image is passed through the model and then it is followed by a couple of convolutional layers with the ReLU activation function. We can notice that the image size is reducing, because we use npadded convolutions, which results in the reduction of the overall dimensionality. Apart from the Convolution blocks, we have encoder block on the left side followed by the decoder block on the right side.

The encoder block has a constant reduction of image size with the help of the max-pooling layers of strides 2. We also have repeated convolutional layers with an increasing number of filters in the encoder architecture. Once we reach the decoder aspect, the number of filters in the convolutional layers start to decrease along with a gradual upsampling in the following layers all the way to the top. We also make use of skip connections that connect the previous outputs with the layers in the decoder blocks.

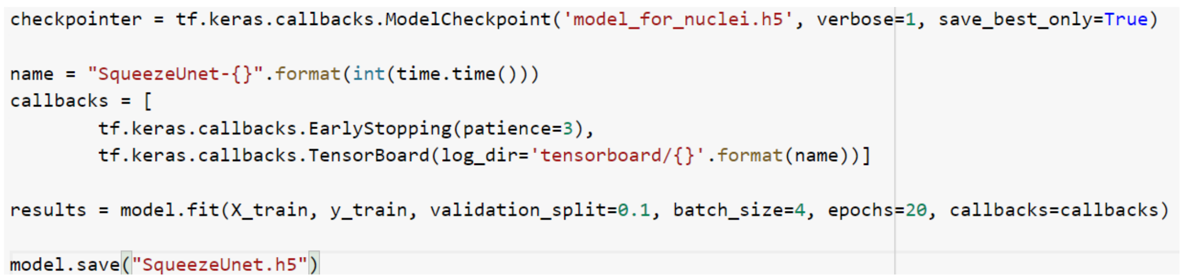
This skip connection is a vital concept to preserve the loss from the previous layers so that they reflect stronger on the overall values. They produce better results and lead to faster model convergence. In the final convolution block, we have a couple of convolutional layers followed by the final convolution layer.

Code snippet:



```
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.utils import array_to_img, img_to_array, load_img

datagen = ImageDataGenerator(
        channel_shift_range=135.0,
        brightness_range=(0.3, 0.8))

image_dir = '/content/Images/Images'
for image_file in image_dir:
  img = load_img(os.path.join(image_dir, image_file))
  x = img_to_array(img)
  x = x.reshape((1,) + x.shape)

  i = 0
  for batch in datagen.flow(x, batch_size=1, save_to_dir='Crop_row_images', save_prefix= image_file, save_format='jpeg'):
      i += 1
      if i > 4:
          break
```

This code uses the ImageDataGenerator class from Keras to perform data augmentation on images. The channel_shift_range argument specifies the range of intensity shifts to be applied to the color channels of the images. In this case, intensity of the color channels will be shifted by a value between -135.0 and 135.0. The brightness_range will be adjusted by a value between 0.3 and 0.8. The code then loads images from the image_dir directory, applies the specified data augmentations to the images, and saves the augmented images to the Crop_row_images directory using the original image file name as a prefix and JPEG format.

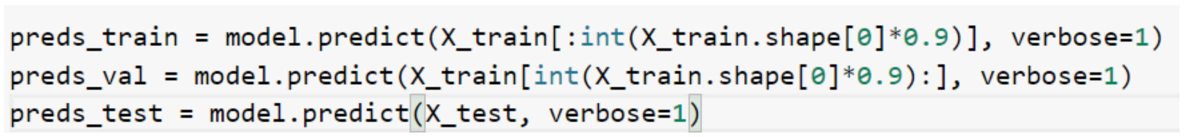


```
checkpointer = tf.keras.callbacks.ModelCheckpoint('model_for_nuclei.h5', verbose=1, save_best_only=True)

name = "SqueezeUnet-{}".format(int(time.time()))
callbacks = [
        tf.keras.callbacks.EarlyStopping(patience=3),
        tf.keras.callbacks.TensorBoard(log_dir='tensorboard/{}'.format(name))]

results = model.fit(X_train, y_train, validation_split=0.1, batch_size=4, epochs=20, callbacks=callbacks)

model.save("SqueezeUnet.h5")
```

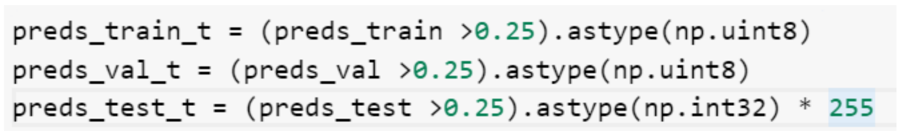
This code trains the SqueezeUNet model using the training data (X_train and y_train). The fit() method is used to train the model with a batch size of 4 and for 20 epochs. The validation_split parameter is set to 0.1, 10% of the training data is used for validation during training. The callbacks parameter is set to a list of two callbacks: EarlyStopping and TensorBoard.

The EarlyStopping callback is used to stop the training if the model's validation loss does not improve for three consecutive epochs. The TensorBoard callback is used to log the model's training progress to a directory named "tensorboard".

After the model is trained, it is saved to a file named "SqueezeUnet.h5" using the save() method. Additionally, the ModelCheckpoint callback is used to save the best model during training to a file named "model_for_nuclei.h5". This callback saves the model with the best validation loss. The verbose parameter is set to 1, which means that progress updates will be printed to the console during training.



```
preds_train = model.predict(X_train[:int(X_train.shape[0]*0.9)], verbose=1)
preds_val = model.predict(X_train[int(X_train.shape[0]*0.9):], verbose=1)
preds_test = model.predict(X_test, verbose=1)
```



```
preds_train_t = (preds_train >0.25).astype(np.uint8)
preds_val_t = (preds_val >0.25).astype(np.uint8)
preds_test_t = (preds_test >0.25).astype(np.int32) * 255
```

preds_train stores -predicts on the first 90% of the training set X_train

preds_val stores- predicts on the last 10% of the training set X_train

preds_test stores - predicts on the test set X_test

- ➔ we are thresholding the predicted probability maps to get the corresponding binary masks. This is done using a threshold value of 0.25.

```
def encode_mask_to_rle(mask):
    '''
    mask: numpy array binary mask
    255 - mask
    0 - background
    Returns encoded run length
    '''
    pixels = mask.flatten()
    pixels = np.concatenate([[0], pixels, [0]])
    runs = np.where(pixels[1:] != pixels[:-1])[0] + 1
    runs[1::2] -= runs[::2]

    return ' '.join(str(x) for x in runs)


def decode_rle_to_mask(rle, height = 240, width = 320):
    '''
    rle : run-length as string formated (start value, count)
    height : height of the mask
    width : width of the mask
    returns binary mask
    '''
    rle = np.array(rle.split(' ')).reshape(-1, 2)
    mask = np.zeros((height*width))
    color = 255
    for i in rle:
        mask[int(i[0]):int(i[0])+int(i[1])] = color

    return mask.reshape(height, width)
```

These are two functions to encode a binary mask as a run-length encoding (RLE) string and decode an RLE string to a binary mask.

The encode_mask_to_rle function takes a binary mask as a numpy array with 255 for the mask and 0 for the background. It then flattens the array and adds a 0 at the beginning and end of the array. The decode_rle_to_mask function takes an RLE string and the height and width of the mask. It first splits the string into pairs of start value and count. Then it creates a numpy array of zeros with the size of the mask and sets the corresponding indices to 255 for each pair of start value and count. The output is a binary mask with 255 for the mask and 0 for the background.

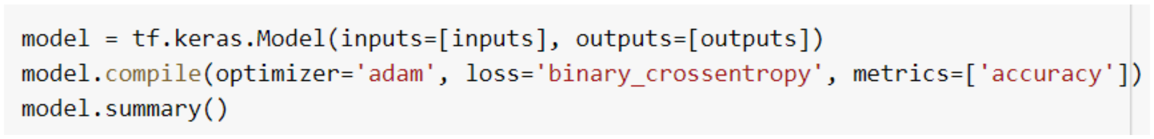
Loss function:

The Binary Cross Entropy (BCE) loss function is commonly used in binary classification problems where the output of the model is a single probability value between 0 and 1. It is defined as:

$$BCE = -(y*\log(p) + (1-y)*\log(1-p))$$

where y is the ground truth label (either 0 or 1), and p is the predicted probability value by the model. The BCE loss function measures the difference between the predicted probability value and the ground truth label.

The Adam optimizer is a popular optimization algorithm used to update the weights of the model during training. It combines the benefits of two other optimization algorithms, Adaptive Moment Estimation (Adam) and Root Mean Square Propagation (RMSProp). The Adam optimizer computes adaptive learning rates for each parameter by estimating first and second moments of the gradients.



```
model = tf.keras.Model(inputs=[inputs], outputs=[outputs])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

When used together, the U-Net model trained with BCE loss and the Adam optimizer is a common choice for binary segmentation tasks, where the output of the model is a binary mask indicating the presence or absence of a certain feature in the input image.

Results and Conclusion:

I trained the network for 25 epochs, and mIoU (mean Intersection over Union) is used as evaluation metric for image segmentation tasks(crop row detection). It measures the degree of overlap between the predicted crop rows and the actual (ground truth) crop rows. The mIoU ranges between 0 and 1, with higher values indicating better model performance. This evaluation is performed by splitting the dataset into training and validation sets.

I got mIoU of 0.21 indicates moderate performance of the crop row detection model. It means that the predicted crop rows have very little overlap with the actual (ground truth) crop rows.