Assessing the performance of a U-net instance segmentation model for ecological application

Introduction

Owing to the mysticism that often surrounds the subject, "Deep Learning" is a relatively misunderstood technology that has been used for quite innovative applications in Biology within the past half-decade. From identifying cancerous cells from biopsy images that previously went undetected (Shen et al., 2019), to accurately predicting the multi-structural composition of a protein in 3D-space from just its amino-acid sequence (Jumper et al., 2021), these powerful tools in Biology (which span research and industry) owe their existence to a class of algorithms that, at their core, simply excel at recognizing "complex motifs from large sequence data sets" (Rusk, 2016).

One class of those "large sequence data sets" are images. As most people are well-aware, one of, if not the most important senses animals possess is that of sight, as over half of the mammalian brain is dedicated to dealing with the plethora of visual imagery we absorb throughout our lifetimes. This task is made difficult not so much by the process of achieving visual acuity, but interpreting the "meaning" represented by a given image. As in, what objects the image contains, how large or small are those objects are, what their relationship to each other is in 3D-space, etc.

Within ecology and evolution, specifically taxonomy, several computer vision applications have recently been built and made available for public use. Ben Weinstein's DeepMeerkat is one such example, training a model to identify specific species of animal within a set of motion-video, which allows for field-researchers to quickly isolate portions of video containing their species of interest without having to manually search through all of their footage (Weinstein, 2018). iNaturalist, the popular biodiversity social network, has built a large database of specimen images along with temporal and location data, all of which has been used to train several highly-accurate models to identify animals down to their genus and species (Timm et al., 2018) Apart from these relatively basic identification tools, however, computer vision has not often been used to deduce patterns in images beyond those that can be identified by the human eye.

This project, then, is specifically concerned with developing a computer-vision model that can be used to detect complex visual motifs in a given organism that may be later used as previously undetectable traits for phylogenetic analysis. One such class of computer-vision model that I hypothesize will be useful for this task is "instance segmentation," which involves a model that can identify which parts of an image, down to the pixel, comprise an object-of-interest (Ajmal et al., 2018). In our case, that object-of-interest would be a given organism, and I hypothesize that elements of the feature-map generated by an instance segmentation model could be used in place of, or in addition to, simplistic, single-feature taxonomic traits to develop higher-resolution phylogenies. Since developing a brand new method of phylogenetic analysis is, in my estimate, outside the scope of this course, I will use this project to train an instance segmentation model on an arbitrary set of animal images that can be used as a stepping-stone to eventually developing this methodology.

The specific model I intend to train is the U-net instance segmentation model. This model was developed at the University of Freiburg in Germany, and was designed to quickly achieve instance segmentation using less than 20 training images. The primary application of this model is instance segmentation of biomedical-specific images, and excels at training on so few because of the higher cost of producing microscopic images (Ronneverger et al., 2015) I am specifically interested in how the model will perform when trained on 3D images of wildlife, as opposed to 2D EM images.

Methods

The data I used to train the U-Net model is available on Kaggle, a prominent data science community and source of a number of datasets available for public use. This particular dataset contains 1210 images of Red Sea reef fish (Fig 1a), along with 1210 manually-annotated masks: 1 for each image (Fig 1b).

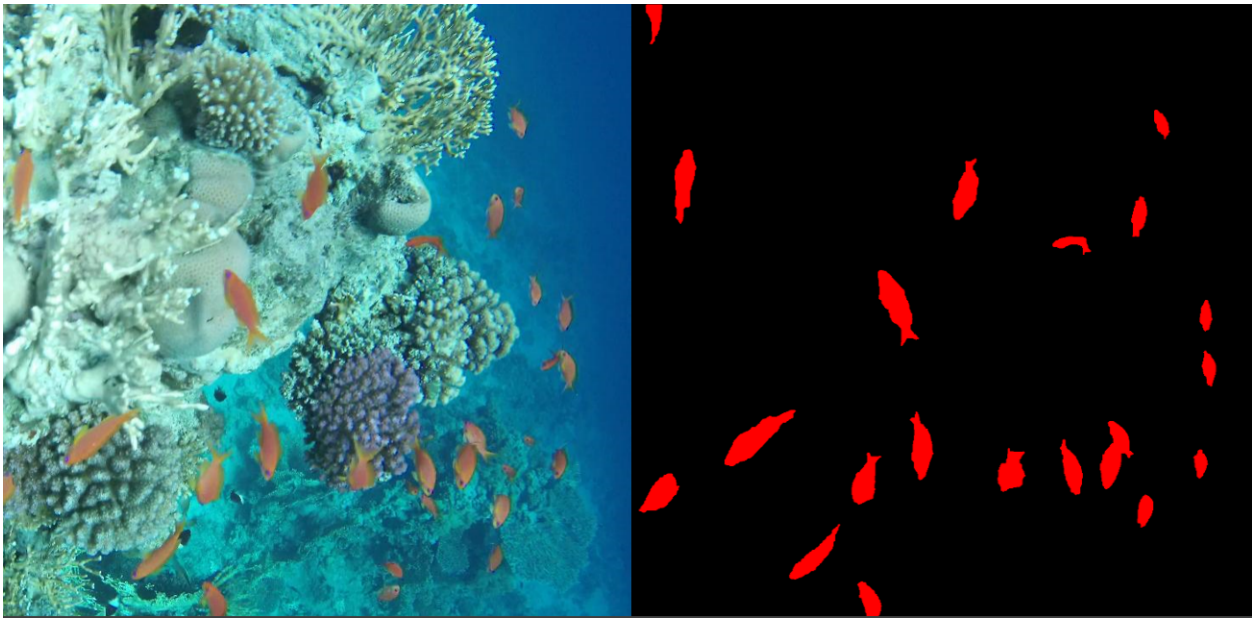

Figure 1: Figure 1

Figure 1. A) A sample input image used to train the U-Net. B) A mask of the same image. This represents the ground-truth value used to train the U-Net, and its ideal output given the input image

In order to satisfy the project requirement of developing within the R language, I used the U-net implementation and guide published by the RStudio AI Blog. Since it was published in 2019, some modification of this implementation was required, and these changes are documented in the code. This implementation of U-net was built with Tensorflow 2.0 Keras, a popular wrapper library for Tensorflow that allows for quick experimentation through the use of modular "plug-and-play" network layers that can be replaced and modified on the fly. It also uses Tensorflow for R and r-reticulate, both of which act as R wrappers for Python 3. Installation of conda is necessary for both the Tensorflow and Python libraries to be installed in a virtual environment, which can get messy if one loses track of the environment which is automatically created by r-reticulate. For that reason, more specific instructions on how to run this model are included in Apendix A - Installation Instructions.

The actual code involved in training and using the U-Net model is not long at all, about 100 lines in total, so I will include it below in several blocks, and comment on the specific function of each block.

Training the U-Net model (R Code)

```
model <- unet(input_shape = c(128, 128, 3))
```

The above code creates a U-Net, whose dimensions have been pre-selected by the Keras implementation available here. This specifies the dimensions of the input images, as well as the number of color channels (RGB). The U-Net will expect these of images it trains on, but also those that it recieves as input for mask prediction.

```
data <- tibble(
  img = list.files(here::here("images"), full.names = TRUE),
  mask = list.files(here::here("mask"), full.names = TRUE)
)
data <- initial_split(data, prop = 0.8)
```

I then load the image and mask file names into a tibble, and split the data between a training and testing set. This will set some data aside for training, and set 20% of the images aside for the model to test its accuracy.

```
training_dataset <- training(data) %>%
  tensor_slices_dataset() %>%
  dataset_map(~.x %>% list_modify(
    img = tf$image$decode_jpeg(tf$io$read_file(.x$img)),
    mask = tf$image$decode_jpeg(tf$io$read_file(.x$mask))
  ))
training_dataset <- training_dataset %>%
  dataset_map(~.x %>% list_modify(
    img = tf$image$convert_image_dtype(.x$img, dtype = tf$float32),
    mask = tf$image$convert_image_dtype(.x$mask, dtype = tf$float32)
  ))
training_dataset <- training_dataset %>%
  dataset_map(~.x %>% list_modify(
    img = tf$image$resize(.x$img, size = shape(128, 128)),
    mask = tf$image$resize(.x$mask, size = shape(128, 128))
  ))
```

I then perform preprocessing of the images, which involves converting reading them in as jpeg files, converting them to their 32-bit floating-point representations, and downscaling them to 128x128 pixels, the input that the U-Net is expecting.

```
#Data augmentation
random_bsh <- function(img) {
  img %>%
    tf$image$random_brightness(max_delta = 0.3) %>%
    tf$image$random_contrast(lower = 0.5, upper = 0.7) %>%
    tf$image$random_saturation(lower = 0.5, upper = 0.7) %>%
    tf$clip_by_value(0, 1)
}
training_dataset <- training_dataset %>%
  dataset_map(~.x %>% list_modify(
    img = random_bsh(.x$img)
  ))
```

Then data-augmentation is performed on the images. This method will randomly modify non-essential values of the image, like brightness, contrast, and saturation. This is done to control for these variables, such that if an input image varied in any of these dimensions, it should not effect the prediction of the U-Net model.

```
model %>% compile(
  optimizer = optimizer_rmsprop(learning_rate = 1e5),
  loss = "binary_crossentropy",
)
model %>% fit(
  training_dataset,
  epochs = 1,
  validation_data = validation_dataset
)
```

It's in this final step where the model is actually trained. It is first fitted with an optimizer (RMSprop, a popular gradient-descent optimizer), and a loss function. Binary Crossentropy will calculate loss based on the difference between the true value of a given pixel of the input image (mask or no mask). The "epochs" variable refers to the number of times the model is trained on the entire set of training images, and in this example is set to 1, which means the training will take less than 10 minutes.

Results

In it's current state, the U-Net model trained on these images does not output any mask, but still reports high accuracy scores when evaluated on the test dataset (Figure 2).

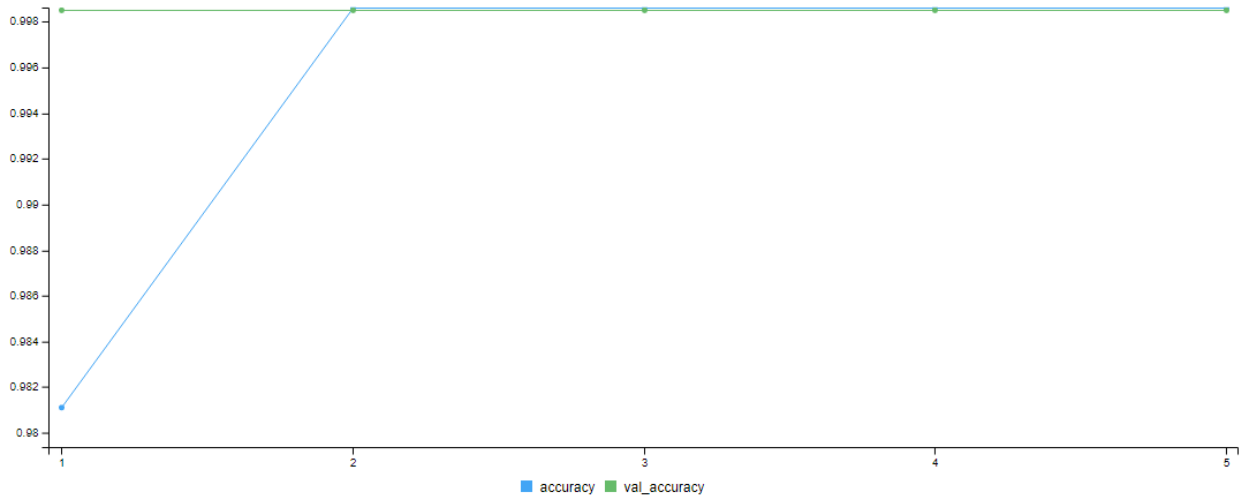

Figure 2. Accuracy plot over 5 epochs of training. "Val_accuracy" refers to the accuracy of the validation set, that which is used for the model to check against as its training, as opposed to the testing set, which it won't see until evaluation.

Discussion

Unfortunately, the current implementation of the U-Net model in this report is non-functional. In this section, I will discuss possible explanations for this, as well as methods that may be useful in the future for obtaining a working instance segmentation model with this dataset.

The most obvious explanation for this model's failure may be the implementation itself, as opposed to the dataset or any particular model layer. Since it was compiled for a previous version of R, with a prior version of Tensorflow 2 for R, it may simply be a case of conflicting package versions and deprecated codebases that inhibited the model. This could be resolved by recreating the U-Net model from scratch in the newest version of Keras, in the newest version of R. This implementation of the U-Net model also had some limitations that were inherent to the organization of the Keras layers. To expedite the training process, this implementation only accepted images that were 128x128 pixels large. This is a significantly reduced quality given the original resolution of the input images (1280 x 720). If I were to redesign the model itself, I would sacrifice the speed of training to allow the input images to remain at their original resolution.

Since configuring a neural network, even with a wrapper library like Keras, is significantly more difficult than implementing one, so it may be more useful to assess the dataset itself for possible complications. There are a few reason this particular dataset may not have yielded a working model. For one, the size and shape of the masks in this dataset are significantly different from those used in the original U-Net paper. While those images were 2D, the images of fish in our dataset were taken from multiple angles, and at a number of distances. When resizing to 128x128, it's likely that these smaller masks did not retain their original shapes, and may even have been lost at such a small resolution.

It's also the case that this dataset did not fully annotate the images of fish in most input/mask image pairs. In Figure 1, for instance, a number of fish in the input image can be observed that were not labeled in the mask. I suspect that this made the task of segmenting a given fish incredibly difficult, since the model would have considered non-annotated fish as background imagery that could be considered distinct from the content behind the masks, which is clearly false. It seems that the largest source of inaccuracy from this model is hidden within a poorly annotated dataset, that if properly compiled, could yield more promising results.

As for future implementations of a segmentation model that would yield a feature map of traits useful for phylogenetic analysis, I suggest a few approaches that may accomplish this in the future. The first of which would be to develop solely in python. Packages such as r-reticulate and Tensorflow for R may be useful for R language natives to quickly access deep learning libraries, but they are highly inefficient on two fronts. The first has to do with the availability of deep learning resources in R. Even with wrapper libraries for several python modules exist, the deep learning ecosystem in python is so much larger than the set of all R wrappers. In addition, R users have to wait for the newest python versions/libraries to have wrappers written for them, or even worse, are forced to write them themselves. Exportability is also a much better-handled feature in Python, with IPython notebooks allowing for a combination of markdown and Python that can be hosted in the cloud through services like Google Colab. The second limitation has to do with the R wrappers themselves. From a pure principles of programming standpoint, the more wrappers that are written for a given function, the more points of potential failure that function has, as may have been demonstrated by this very project.

More specifically, a future python implementation might also include a model (U-Net or otherwise) that was first trained on a very large general image dataset before training on images of a specific organism. This is referred to as "transfer learning," and it involves creating a prior map of features that may be consistent across all images before specifying those of a particular object of interest. While it may be difficult to separate the feature spaces of these initial maps from those used for phylogenetic traits, it is a method that will certainly improve the accuracy of a segmentation model, and is worth acknowledging.

References

Ajmal, H., Rehman, S., Farooq, U., Ain, Q. U., Riaz, F., & Hassan, A. (2018, April). Convolutional neural network based image segmentation: a review. In Pattern Recognition and Tracking XXIX (Vol. 10649, p. 106490N). International Society for Optics and Photonics.

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., ... & Hassabis, D. (2021). Highly accurate protein structure prediction with AlphaFold. Nature, 596(7873), 583-589.

Ronneberger, O., Fischer, P., & Brox, T. (2015, October). U-net: Convolutional networks for biomedical image segmentation. In International Conference on Medical image computing and computer-assisted intervention (pp. 234-241). Springer, Cham.

Rusk, N. (2016). Deep learning. Nature Methods, 13(1), 35-35.

Shen, L., Margolies, L. R., Rothstein, J. H., Fluder, E., McBride, R., & Sieh, W. (2019). Deep learning to improve breast cancer detection on screening mammography. Scientific reports, 9(1), 1-12.

Timm, M., Maji, S., & Fuller, T. (2018). Large-scale ecological analyses of animals in the wild using computer vision. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (pp. 1896-1898).

Weinstein, B. G. (2018). A computer vision for animal ecology. Journal of Animal Ecology, 87(3), 533-545.

Apendix A - Installing Tensorflow and Configuring Miniconda

Installing Tensorflow Keras and navigating its respective environment can be difficult for first-time users. Here are a few pointers for configuring an environement that is compatible with the code in this repository.

1. When first running the code in this repository in a fresh R Studio session, the installation of r-tensorflow/unet will install r-reticulate, the Python 3 wrapper. This will also prompt the instillation of Miniconda, and create an environment called "r-reticulate" where a version of Python 3 will be installed, regardless of if it already on your machine. This environment can be accessed through the bash terminal by typing "conda activate r-reticulate," or graphically with the Anaconda application.

2. If Tensorflow for R says that you are running a 32-bit version of python that is incompatible with Tensorflow, it may be necessary to reinstall the latest 64-bit version of python in the r-reticulate environment.

3. Install the Keras package using the following:

```
install.packages("keras")
```

After loading the Keras package, the following function must be run:

```
install_keras()
```

This will install the proper version of Tensorflow to the conda environment you're currently in (r-reticulate). You can make sure this has been done by seeing if the boxes next to Tensorflow and Keras are checked in the Packages tab, and that they have the same version number.

4. The images used to train U-Net are too large to host on GitHub. Here is their Kaggle link. The image folder file-paths are currently directed at folders within the project directory, so you can simply empty the contents of the 720p folder into the "images" folder, and the contents of the masks folder into the "mask" folder in the current project.