Spring 2020 Course Project

By Liam Shalon, Haiyu Wan, and Zexi Liu

Github is here:
https://github.com/lshalon/CSE514CourseProject/blob/master/caching_and_preprocessing.py
Our master Colab notebook can be found here:
https://colab.research.google.com/drive/11JLI2fE8kmnyBuWY2xcymREq9SLAx9Gp

We highly recommend that you look through our Colab notebook to supplement your readings in this paper.

Introduction

In computer vision, image segmentation is the process of partitioning a digital image into multiple segments (sets of pixels, also known as image objects). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze. Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics. In this project we will use the data form Pascal voc 2012 and compare FCN, Unet, ResUnet and PSPNet based on the dataset. Based on the comparison result, we decide to choose the Unet to do further research. In the second section, We try to improve the optimizer, loss function and metrics to get a better performance of our model. At the last part of our project we try to reduce the cost of our model and keep the performance to make it more convenient for the users.

Section 1 - Project setup, model space defined

1.1 Project setup

The Pascal dataset has naturally a few different splits. For this project, we use the built-in splits provided by the dataset.

The validation has 1449 samples and the training dataset has 2913 samples.

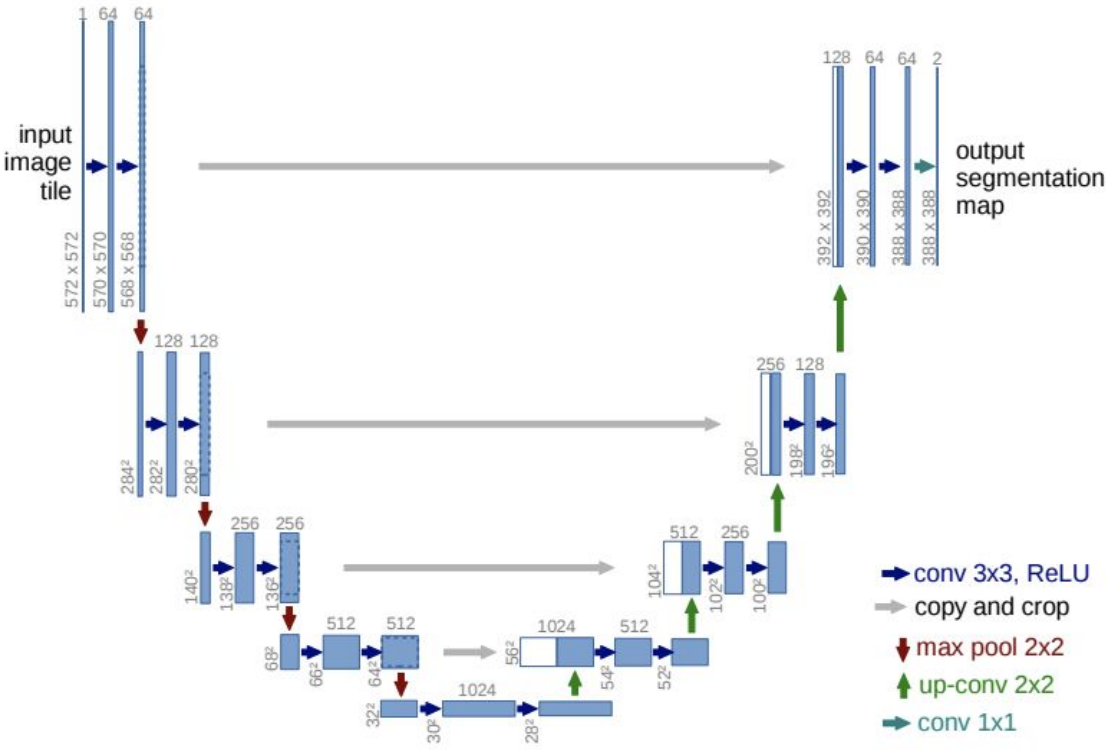
We use the same seed in our generator to get the same shuffling across different runs.

Furthermore, we are caching the labels in our notebook so that we don't have to make conversions constantly to allow for faster training time.

1.2 Model selection

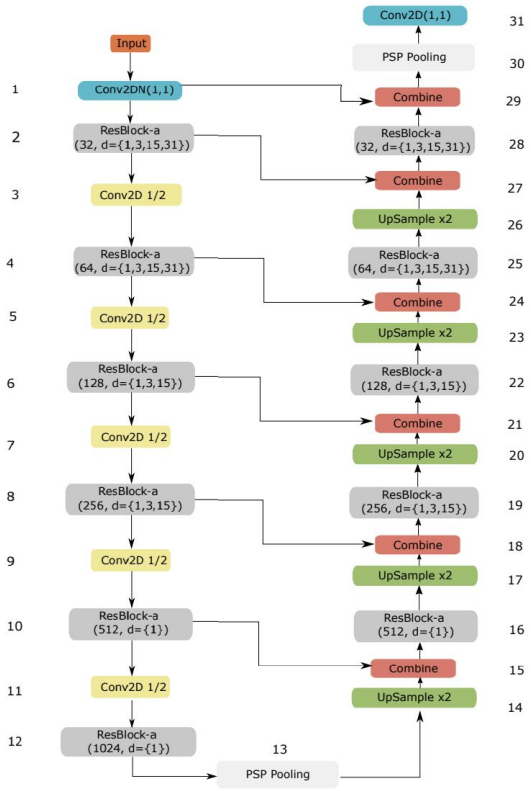
Below is a breakdown of the different general model architectures that we test.

UNET:



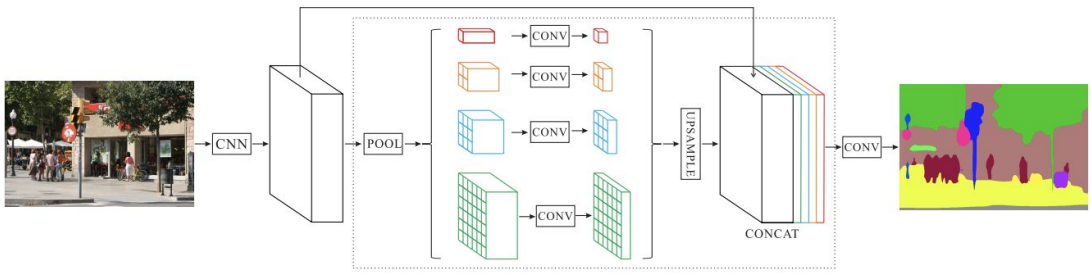
U-net architecture. Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

ResUnet:



Our ResUnet network is a simple version this Overview.The left (downward) branch is the encoder part of the architecture. The right (upward) branch is the decoder. The last convolutional layer has as many channels as there are distinct classes.

PSPNet:



The third model we tried here is the PSPNet, which refers to Pyramid Scene Parsing Network. Compared with FCN, it can resolve several issues for complex scene parsing. Context relationship is universal and important especially for complex scene understanding.
In FCN, lack of the ability to collect contextual information increases the chance of misclassification. Given an input image (a), we first use CNN to get the feature map of the last convolutional layer (b), then a pyramid parsing module is applied to harvest different sub-region representations, followed by

upsampling and concatenation layers to form the final feature representation, which carries both local and global context information in (c). Finally, the representation is fed into a convolution layer to get the final per-pixel prediction (d).

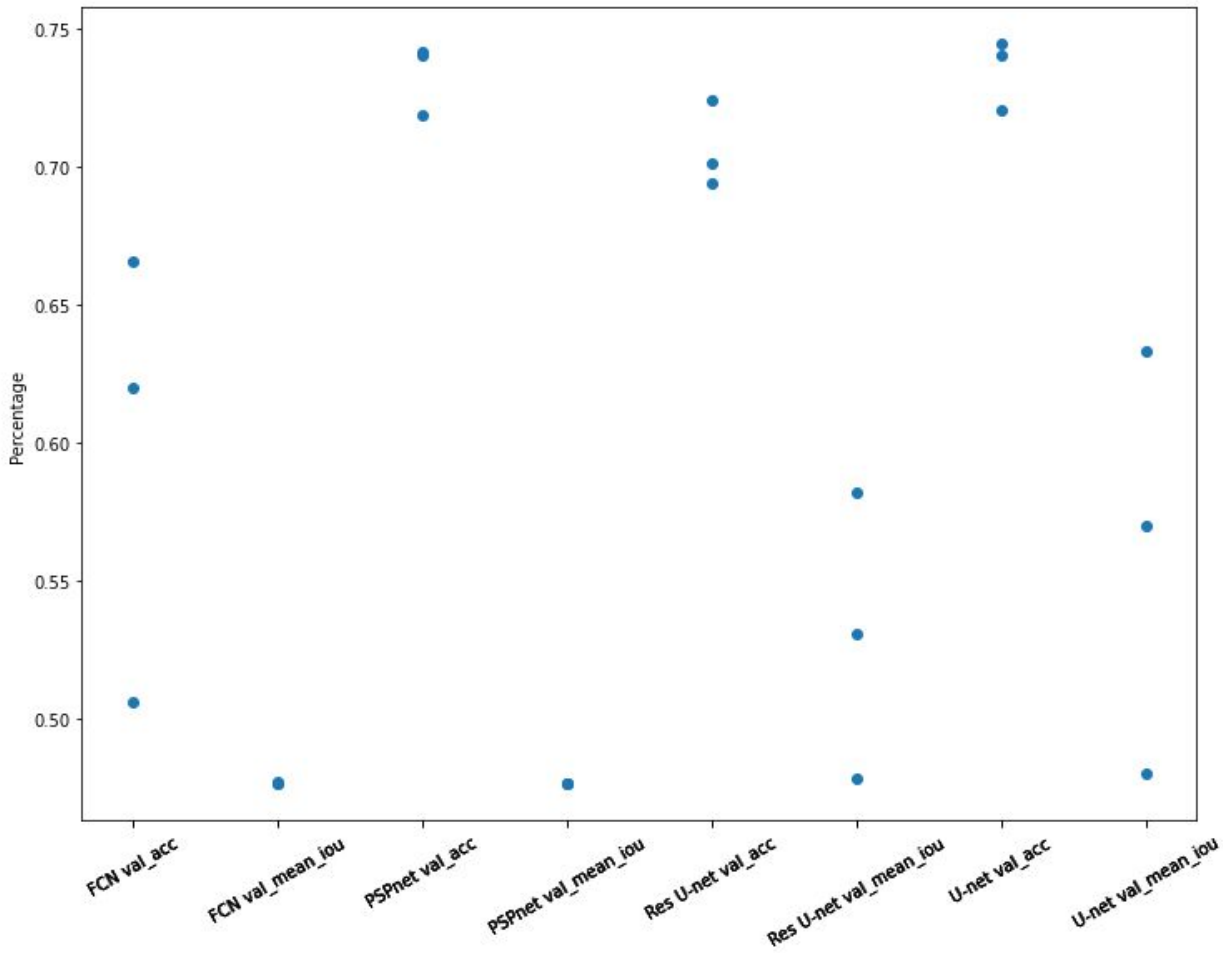
FCN:

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 128, 128, 3)]	0
block1_conv1 (Conv2D)	(None, 128, 128, 64)	1792
block1_conv2 (Conv2D)	(None, 128, 128, 64)	36928
block1_pool (MaxPooling2D)	(None, 64, 64, 64)	0
block2_conv1 (Conv2D)	(None, 64, 64, 128)	73856
block2_conv2 (Conv2D)	(None, 64, 64, 128)	147584
block2_pool (MaxPooling2D)	(None, 32, 32, 128)	0
block3_conv1 (Conv2D)	(None, 32, 32, 256)	295168
block3_conv2 (Conv2D)	(None, 32, 32, 256)	590080
block3_conv3 (Conv2D)	(None, 32, 32, 256)	590080
block3_pool (MaxPooling2D)	(None, 16, 16, 256)	0
block4_conv1 (Conv2D)	(None, 16, 16, 512)	1180160
block4_conv2 (Conv2D)	(None, 16, 16, 512)	2359808
block4_conv3 (Conv2D)	(None, 16, 16, 512)	2359808
block4_pool (MaxPooling2D)	(None, 8, 8, 512)	0
fc1 (Conv2D)	(None, 8, 8, 1024)	25691136
dropout_4 (Dropout)	(None, 8, 8, 1024)	0
fc2 (Conv2D)	(None, 8, 8, 1024)	1049600
dropout_5 (Dropout)	(None, 8, 8, 1024)	0
conv2d_2 (Conv2D)	(None, 8, 8, 21)	21525
up_sampling2d_2 (UpSampling2	(None, 128, 128, 21)	0

Total params: 34,397,525
Trainable params: 34,397,525
Non-trainable params: 0

The other model that we are trying is a fully convolutional network. As mentioned in the PSPNet section, there are many downfalls of FCN.

We tested the four different models and compared Mean IOUs and accuracies. Training each with different learning rates that we tweaked to get the most optimal performance. 3 independent runs, and we took the maximum value of both mean iou and accuracy to serve as that run's point.



We ended up focusing on the U-net model.

1.3 Model implementation

```
#building U-NET
def unet():
  inputs = Input((128, 128, 3))

  c1 = Conv2D(64,(3, 3), activation='relu', kernel_initializer='he_normal', padding='same') (inputs)
  c1 = BatchNormalization()(c1)
  c1 = Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same') (c1)
  c1 = BatchNormalization()(c1)
  p1 = MaxPooling2D((2, 2)) (c1)

  c2 = Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same') (p1)
  c2 = BatchNormalization()(c2)
  c2 = Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same') (c2)
  c2 = BatchNormalization()(c2)
  p2 = MaxPooling2D((2, 2)) (c2)

  c3 = Conv2D(256, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same') (p2)
  c3 = BatchNormalization()(c3)
  c3 = Conv2D(256, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same') (c3)
  c3 = BatchNormalization()(c3)
  p3 = MaxPooling2D((2, 2)) (c3)

  c4 = Conv2D(512, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same') (p3)
  c4 = BatchNormalization()(c4)
  c4 = Conv2D(512, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same') (c4)
  c4 = BatchNormalization()(c4)
  p4 = MaxPooling2D(pool_size=(2, 2)) (c4)

  c5 = Conv2D(1024, (3, 3), activation='relu', padding='same') (p4)
  c5 = BatchNormalization()(c5)
  c5 = Conv2D(1024, (3, 3), activation='relu',  padding='same') (c5)
  c5 = BatchNormalization()(c5)

  u6 = Conv2DTranspose(512, (2, 2), strides=(2, 2), padding='same') (c5)
  u6 = concatenate([u6, c4])
  c6 = Conv2D(512, (3, 3), activation='relu',  padding='same') (u6)
  c6 = Conv2D(512, (3, 3), activation='relu',  padding='same') (c6)

  u7 = Conv2DTranspose(256, (2, 2), strides=(2, 2), padding='same') (c6)
  u7 = concatenate([u7, c3])
  c7 = Conv2D(256, (3, 3), activation='relu',  padding='same') (u7)
  c7 = Conv2D(256, (3, 3), activation='relu',  padding='same') (c7)

  u8 = Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same') (c7)
  u8 = concatenate([u8, c2])
  c8 = Conv2D(128, (3, 3), activation='relu',  padding='same') (u8)
  c8 = Conv2D(128,(3, 3), activation='relu', padding='same') (c8)

  u9 = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same') (c8)
  u9 = concatenate([u9, c1], axis=3)
  c9 = Conv2D(64,(3, 3), activation='relu',  padding='same') (u9)
  c9 = Conv2D(64,(3, 3), activation='relu',  padding='same') (c9)

  outputs = Conv2D(21, (1, 1), activation='softmax') (c9)

  model = Model(inputs=[inputs], outputs=[outputs])

  return model
```

This is our U-net implementation. You can find more of this in our notebook.

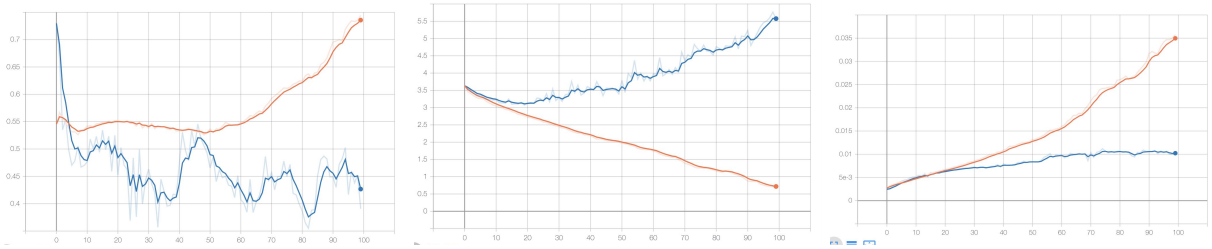
Section 2 - Parameter optimization

2.1 Optimizer optimization

----test -----train

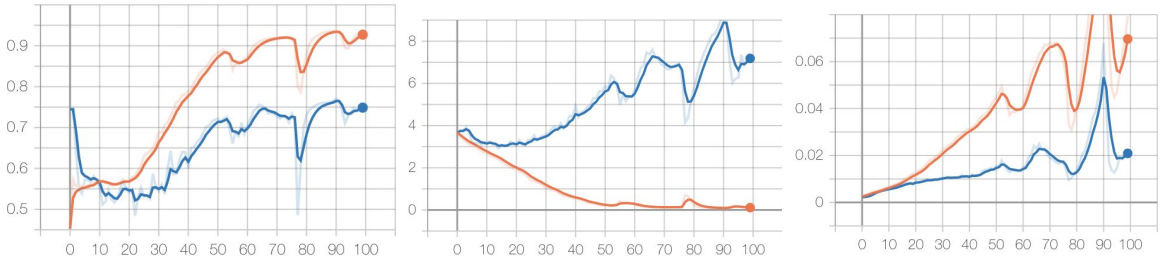
Optimizer Adam(Learning rate:1e-3):

Acurracy: Loss: mean_iou:



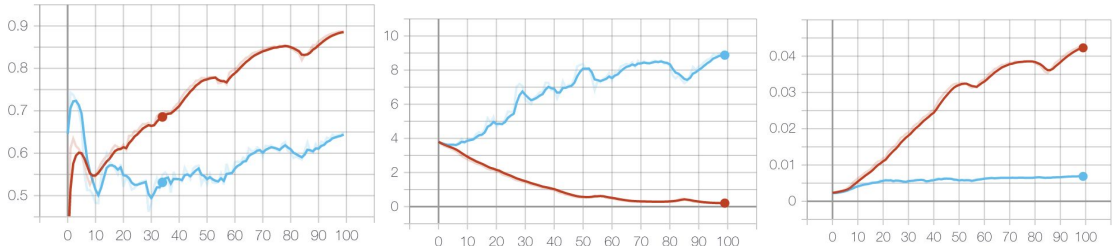
Adam(Learning rate:1e-4):

Accuracy: Loss: mean_iou:



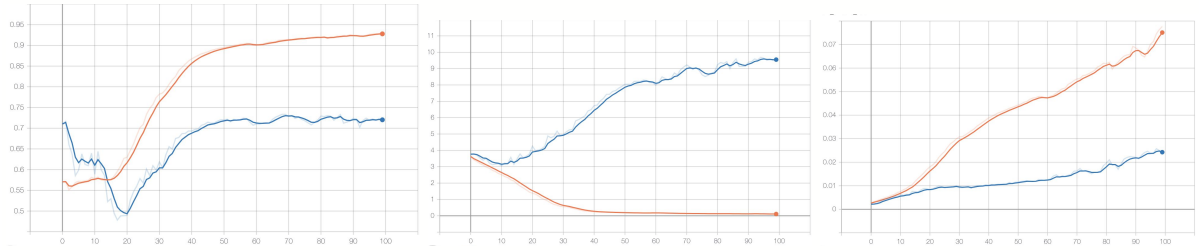
Adam(Learning rate:1e-5):

Accuracy: Loss: Mean_iou:



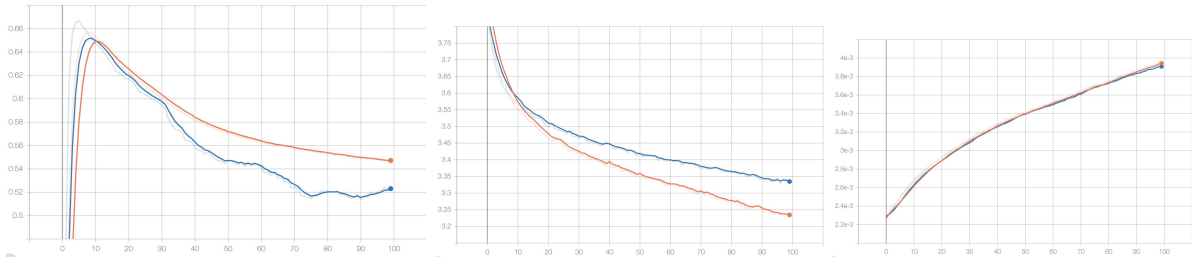
Adam(Learning rate=1e-4, decay=1e-3):

Accuracy: Loss: Mean_iou:



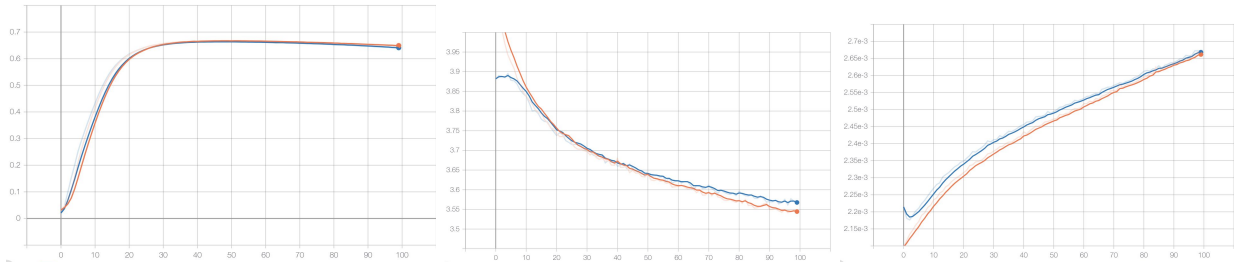
SGD(Learning rate =1e-3):

Accuracy: Loss: Mean_iou:



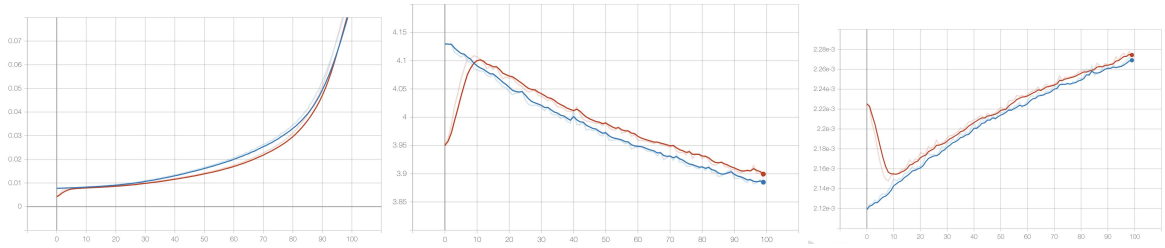
SGD(Learning rate = 1e-4)

Accuracy: Loss: Mean_iou:



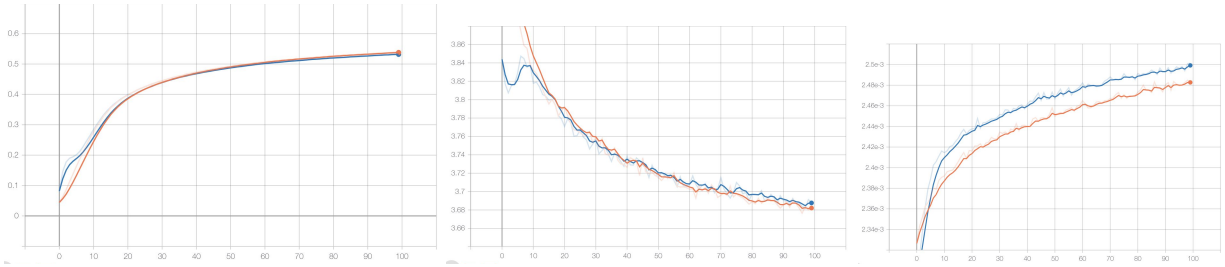
SGD(Learning rate = 1e-5):

Accuracy: Loss: Mean_iou:



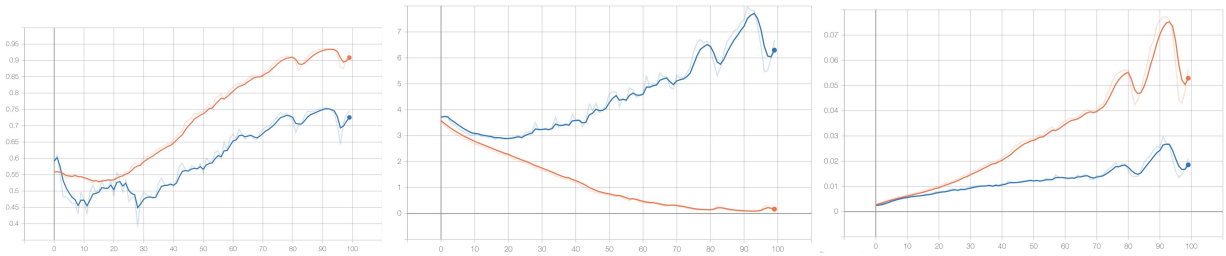
SGD(learning rate = 1e-4, decay = 1e-3)

Accuracy: Loss: Mean_iou:



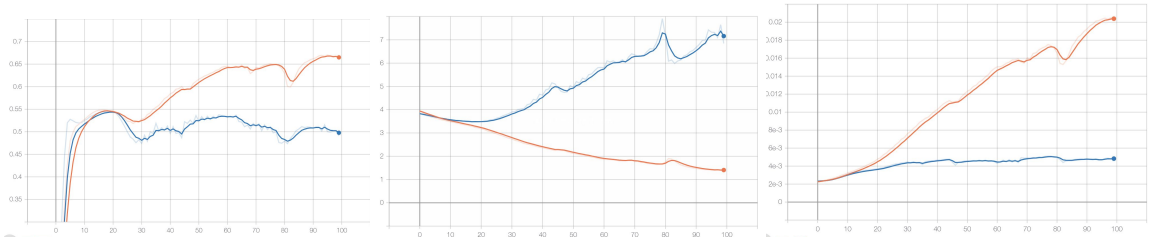
Adamax(Learning rate = 1e-3):

Accuracy: Loss: Mean_iou:



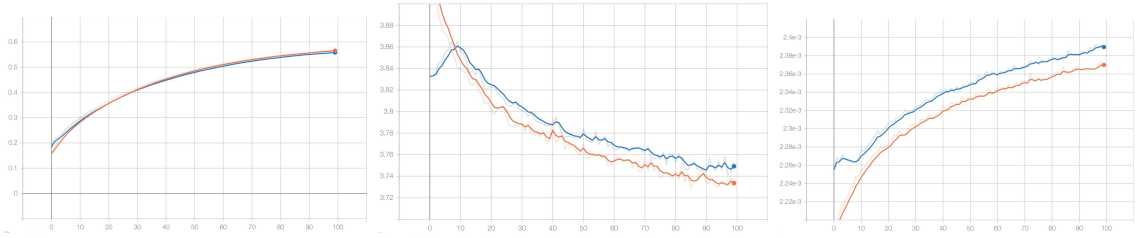
Adamax(Learning rate =1e-4):

Accuracy: Loss: Mean_iou:



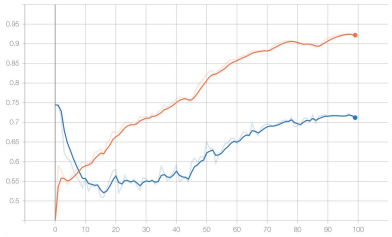
AdAmax(Learning rate = 1e-5):

Accuracy: Loss: Mean_iou:

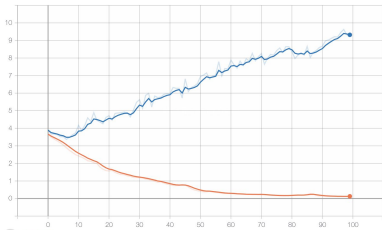


Adamax(Learning rate = 1e-4, decay=1e-3):

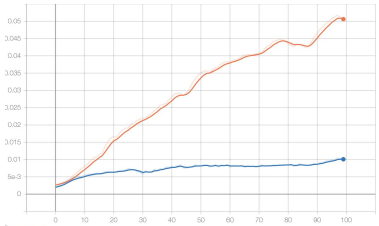
Accuracy:



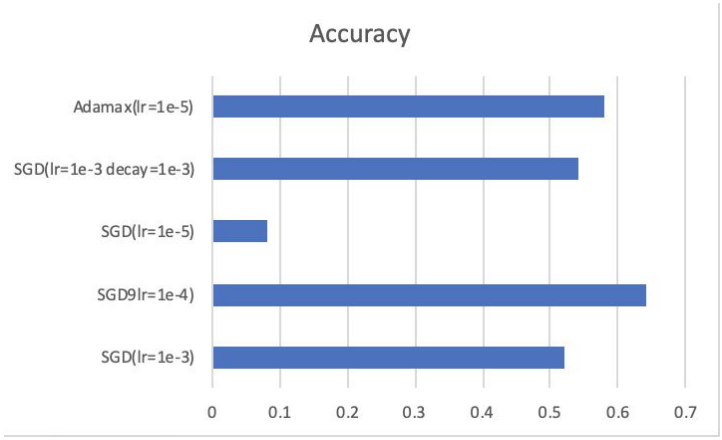
Loss:

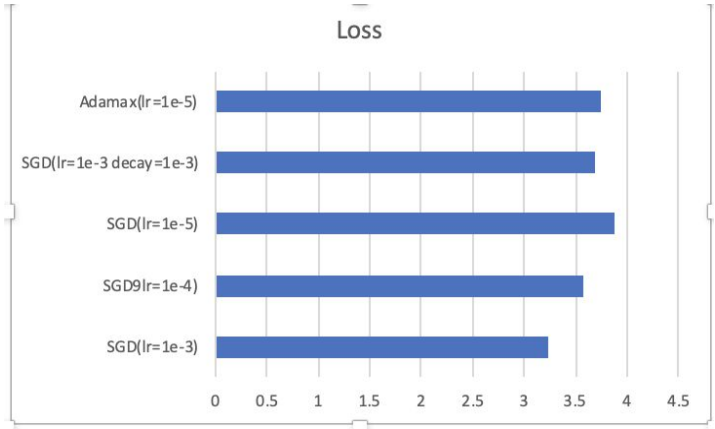


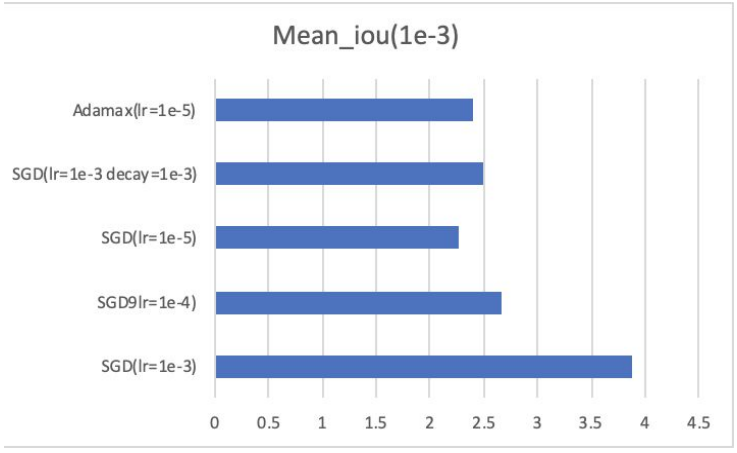
Mean_iou:



(mean_iou, accuracy, Loss comparison of converged result)







For the optimizer selection we choose the Adam, SGD and Adamax with learning rate: 1e-3, 1e-4, 1e-5, (1e-3, decay=1e-3).
Adam:
Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum. It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum.

SGD:
Stochastic gradient descent is an iterative method for optimizing an objective function. Instead of using the actual gradient, SGD use stochastic approximation of gradient descent. It is calculated from a randomly selected subset of the data. SGD in big data applications can reduce the computational burden.

Adamax:
It is a variant of Adam based on the infinity norm. Adamax is sometimes superior to adam, especially in models with embedding. It changes the v factor in the adam update rule scales the gradient inversely proportionally to norm of the past gradients and current gradient $|g|^2$.

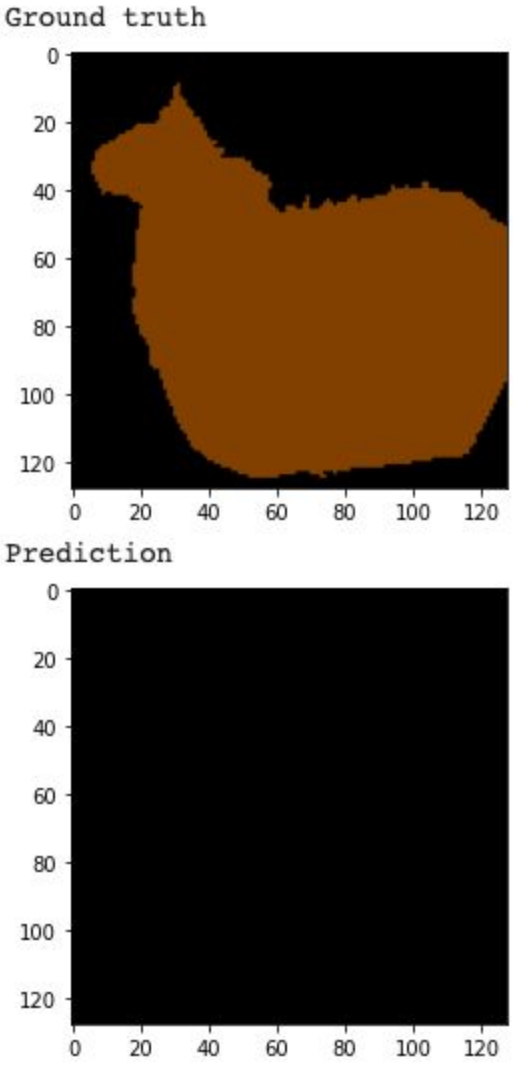
Mean_iou:
Pixel accuracy is easiest to understand but it is not a good metric for the image segmentation.
You may get a very high accuracy if you predict all the pictures as background. This issue is called class imbalance. To get a more accurate metric, we design a custom mean_iou metric.
The Intersection-Over-Union is one of the most commonly used metrics on semantic segmentation.
Mean Iou = (classes' accuracy +background's accuracy)/number of classes
Now we can get a more accurate metric. For more, we ignore the background class to improve the evaluation performance of our project.

The result above is based on the Unet model.In the graph, we can see that the SGD with learning rate 1e-3 has the lowest test loss. SGD with learning rate 1e-4 and decay 1e-3 has the best performance. The accuracy, mean_iou grows as we expected and the loss converged. Adam seems to have the worst performance compared with Adamax and SGD. With different learning rates, adam test evaluation has bad performance and seems not to converge . When learning rate equals to 1e-5, the adamax optimizer has the best performance and the learning rate converges.

2.2 Loss function optimization

Initially, when we trained using categorical cross-entropy, we found that our models weren't learning anything.

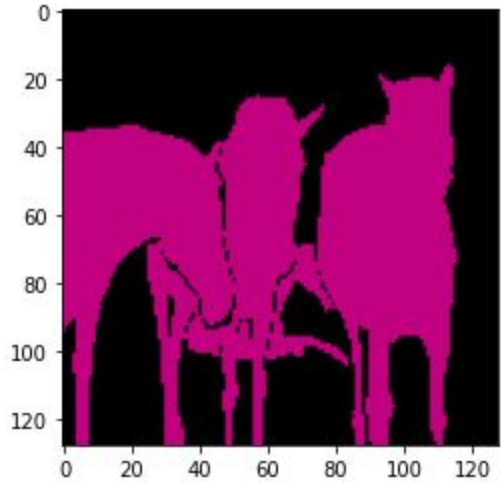
Our predictions looked a lot like this:



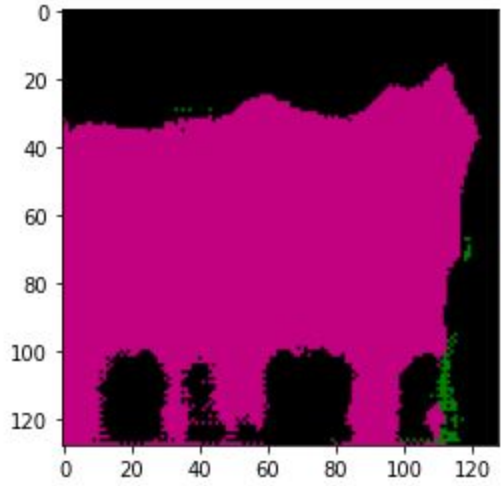
So, we decided to inversely weigh the classes in our categorical cross-entropy loss function as to encourage our model to take risks. The difference was significant.

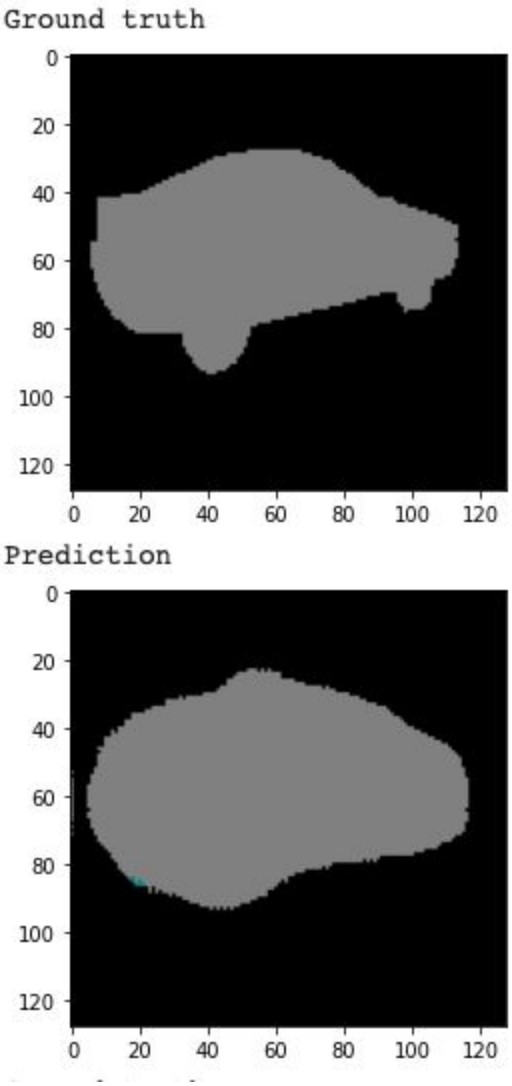
You can see that after 100 epochs, we began to get results that looked a lot like this:

Ground truth



Prediction

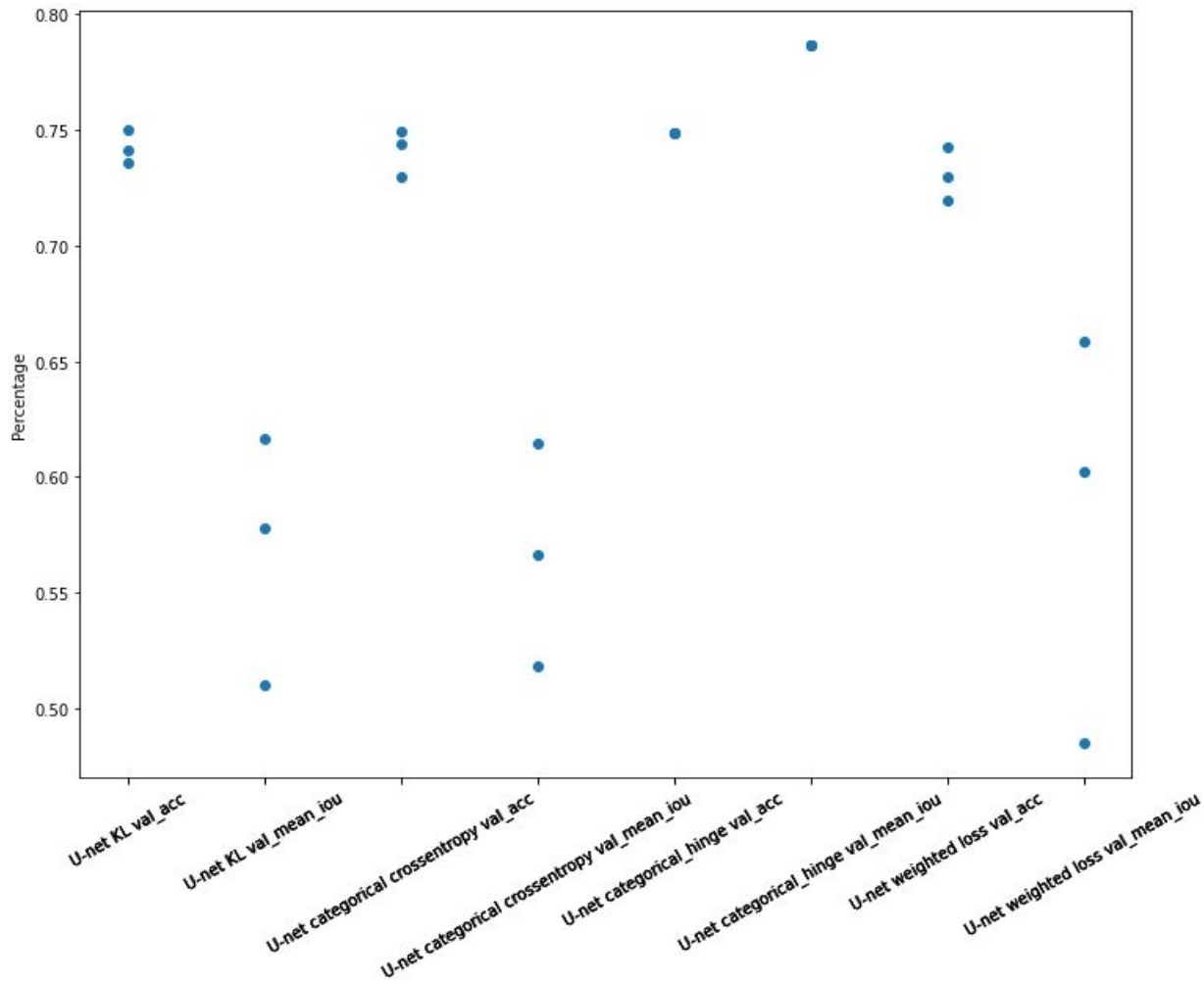




We also decided to go back and as a sanity check test some different loss functions:

All of these are with the adam optimizer, a learning rate of 1e-4, and a decay of 1e-3.

Here were our results after training each with 25 epochs and using the same comparison strategy as before:



Hinge seems to get very consistently high results - perhaps it is worth investigating. Although, since we are only running each trial for 25 epochs, this test is inconclusive. Moreover, we just don't know what the model is predicting - only black backgrounds? As a next step, because we have ruled out the standard cross-entropy loss function, I'm curious to run a weighted experiment on these other loss functions and see which weighted loss function performs best.

Section 3 - Polishing

3.1 Size reduction:

We are testing three different sizes of the U-net…

The original looks like this:

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 128, 128, 3)	0	
conv2d (Conv2D)	(None, 128, 128, 64)	1792	input_1[0][0]
batch_normalization (BatchNorma	(None, 128, 128, 64)	256	conv2d[0][0]
conv2d_1 (Conv2D)	(None, 128, 128, 64)	36928	batch_normalization[0][0]
batch_normalization_1 (BatchNor	(None, 128, 128, 64)	256	conv2d_1[0][0]
max_pooling2d (MaxPooling2D)	(None, 64, 64, 64)	0	batch_normalization_1[0][0]
conv2d_2 (Conv2D)	(None, 64, 64, 128)	73856	max_pooling2d[0][0]
batch_normalization_2 (BatchNor	(None, 64, 64, 128)	512	conv2d_2[0][0]
conv2d_3 (Conv2D)	(None, 64, 64, 128)	147584	batch_normalization_2[0][0]
batch_normalization_3 (BatchNor	(None, 64, 64, 128)	512	conv2d_3[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 32, 32, 128)	0	batch_normalization_3[0][0]
conv2d_4 (Conv2D)	(None, 32, 32, 256)	295168	max_pooling2d_1[0][0]
batch_normalization_4 (BatchNor	(None, 32, 32, 256)	1024	conv2d_4[0][0]
conv2d_5 (Conv2D)	(None, 32, 32, 256)	590080	batch_normalization_4[0][0]
batch_normalization_5 (BatchNor	(None, 32, 32, 256)	1024	conv2d_5[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 16, 16, 256)	0	batch_normalization_5[0][0]
conv2d_6 (Conv2D)	(None, 16, 16, 512)	1180160	max_pooling2d_2[0][0]
batch_normalization_6 (BatchNor	(None, 16, 16, 512)	2048	conv2d_6[0][0]
conv2d_7 (Conv2D)	(None, 16, 16, 512)	2359808	batch_normalization_6[0][0]
batch_normalization_7 (BatchNor	(None, 16, 16, 512)	2048	conv2d_7[0][0]
max_pooling2d_3 (MaxPooling2D)	(None, 8, 8, 512)	0	batch_normalization_7[0][0]
conv2d_8 (Conv2D)	(None, 8, 8, 1024)	4719616	max_pooling2d_3[0][0]
batch_normalization_8 (BatchNor	(None, 8, 8, 1024)	4096	conv2d_8[0][0]
conv2d_9 (Conv2D)	(None, 8, 8, 1024)	9438208	batch_normalization_8[0][0]
batch_normalization_9 (BatchNor	(None, 8, 8, 1024)	4096	conv2d_9[0][0]
conv2d_transpose (Conv2DTranspo	(None, 16, 16, 512)	2097664	batch_normalization_9[0][0]
concatenate (Concatenate)	(None, 16, 16, 1024)	0	conv2d_transpose[0][0] batch_normalization_7[0][0]

```
conv2d_10 (Conv2D)              (None, 16, 16, 512)  4719104     concatenate[0][0]
conv2d_11 (Conv2D)              (None, 16, 16, 512)  2359808     conv2d_10[0][0]
conv2d_transpose_1 (Conv2DTrans (None, 32, 32, 256)  524544      conv2d_11[0][0]
concatenate_1 (Concatenate)     (None, 32, 32, 512)  0           conv2d_transpose_1[0][0]
                                                                 batch_normalization_5[0][0]
conv2d_12 (Conv2D)              (None, 32, 32, 256)  1179904     concatenate_1[0][0]
conv2d_13 (Conv2D)              (None, 32, 32, 256)  590080      conv2d_12[0][0]
conv2d_transpose_2 (Conv2DTrans (None, 64, 64, 128)  131200      conv2d_13[0][0]
concatenate_2 (Concatenate)     (None, 64, 64, 256)  0           conv2d_transpose_2[0][0]
                                                                 batch_normalization_3[0][0]
conv2d_14 (Conv2D)              (None, 64, 64, 128)  295040      concatenate_2[0][0]
conv2d_15 (Conv2D)              (None, 64, 64, 128)  147584      conv2d_14[0][0]
conv2d_transpose_3 (Conv2DTrans (None, 128, 128, 64) 32832       conv2d_15[0][0]
concatenate_3 (Concatenate)     (None, 128, 128, 128 0           conv2d_transpose_3[0][0]
                                                                 batch_normalization_1[0][0]
conv2d_16 (Conv2D)              (None, 128, 128, 64) 73792       concatenate_3[0][0]
conv2d_17 (Conv2D)              (None, 128, 128, 64) 36928       conv2d_16[0][0]
conv2d_18 (Conv2D)              (None, 128, 128, 21) 1365        conv2d_17[0][0]
==================================================================================================
Total params: 31,048,917
Trainable params: 31,040,981
Non-trainable params: 7,936
__________________________________________________________________________________________________
```

A medium-sized version looks like this:

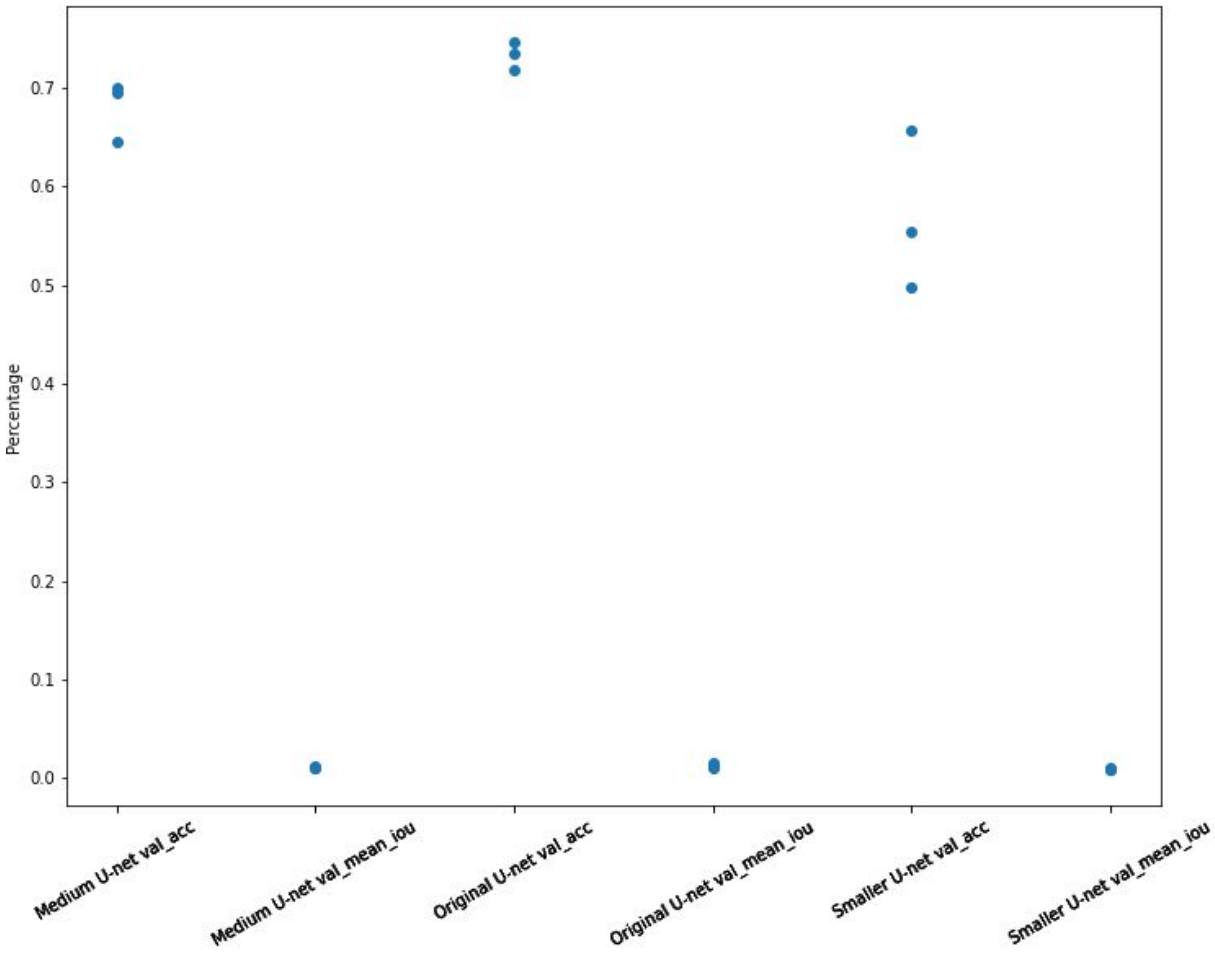
```
Model: "model_1"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #   Connected to
=================================================================================================
input_2 (InputLayer)         [(None, 128, 128, 3)      0
_________________________________________________________________________________________________
conv2d_19 (Conv2D)           (None, 128, 128, 64)      1792      input_2[0][0]
_________________________________________________________________________________________________
batch_normalization_10 (BatchNo (None, 128, 128, 64)   256       conv2d_19[0][0]
_________________________________________________________________________________________________
conv2d_20 (Conv2D)           (None, 128, 128, 64)      36928     batch_normalization_10[0][0]
_________________________________________________________________________________________________
batch_normalization_11 (BatchNo (None, 128, 128, 64)   256       conv2d_20[0][0]
_________________________________________________________________________________________________
max_pooling2d_4 (MaxPooling2D)  (None, 64, 64, 64)     0         batch_normalization_11[0][0]
_________________________________________________________________________________________________
conv2d_21 (Conv2D)           (None, 64, 64, 128)       73856     max_pooling2d_4[0][0]
_________________________________________________________________________________________________
batch_normalization_12 (BatchNo (None, 64, 64, 128)    512       conv2d_21[0][0]
_________________________________________________________________________________________________
conv2d_22 (Conv2D)           (None, 64, 64, 128)       147584    batch_normalization_12[0][0]
_________________________________________________________________________________________________
batch_normalization_13 (BatchNo (None, 64, 64, 128)    512       conv2d_22[0][0]
_________________________________________________________________________________________________
max_pooling2d_5 (MaxPooling2D)  (None, 32, 32, 128)    0         batch_normalization_13[0][0]
_________________________________________________________________________________________________
conv2d_23 (Conv2D)           (None, 32, 32, 256)       295168    max_pooling2d_5[0][0]
_________________________________________________________________________________________________
batch_normalization_14 (BatchNo (None, 32, 32, 256)    1024      conv2d_23[0][0]
_________________________________________________________________________________________________
conv2d_24 (Conv2D)           (None, 32, 32, 256)       590080    batch_normalization_14[0][0]
_________________________________________________________________________________________________
batch_normalization_15 (BatchNo (None, 32, 32, 256)    1024      conv2d_24[0][0]
_________________________________________________________________________________________________
max_pooling2d_6 (MaxPooling2D)  (None, 16, 16, 256)    0         batch_normalization_15[0][0]
_________________________________________________________________________________________________
conv2d_25 (Conv2D)           (None, 16, 16, 512)       1180160   max_pooling2d_6[0][0]
_________________________________________________________________________________________________
batch_normalization_16 (BatchNo (None, 16, 16, 512)    2048      conv2d_25[0][0]
_________________________________________________________________________________________________
conv2d_26 (Conv2D)           (None, 16, 16, 512)       2359808   batch_normalization_16[0][0]
_________________________________________________________________________________________________
batch_normalization_17 (BatchNo (None, 16, 16, 512)    2048      conv2d_26[0][0]
_________________________________________________________________________________________________
conv2d_transpose_4 (Conv2DTrans (None, 32, 32, 256)    524544    batch_normalization_17[0][0]
_________________________________________________________________________________________________
concatenate_4 (Concatenate)  (None, 32, 32, 512)       0         conv2d_transpose_4[0][0]
                                                                 batch_normalization_15[0][0]
_________________________________________________________________________________________________
conv2d_27 (Conv2D)           (None, 32, 32, 256)       1179904   concatenate_4[0][0]
_________________________________________________________________________________________________
conv2d_28 (Conv2D)           (None, 32, 32, 256)       590080    conv2d_27[0][0]
_________________________________________________________________________________________________
conv2d_transpose_5 (Conv2DTrans (None, 64, 64, 128)    131200    conv2d_28[0][0]
_________________________________________________________________________________________________
concatenate_5 (Concatenate)  (None, 64, 64, 256)       0         conv2d_transpose_5[0][0]
                                                                 batch_normalization_13[0][0]
_________________________________________________________________________________________________
conv2d_29 (Conv2D)           (None, 64, 64, 128)       295040    concatenate_5[0][0]
_________________________________________________________________________________________________
conv2d_30 (Conv2D)           (None, 64, 64, 128)       147584    conv2d_29[0][0]
_________________________________________________________________________________________________
conv2d_transpose_6 (Conv2DTrans (None, 128, 128, 64)   32832     conv2d_30[0][0]
_________________________________________________________________________________________________
concatenate_6 (Concatenate)  (None, 128, 128, 128      0         conv2d_transpose_6[0][0]
                                                                 batch_normalization_11[0][0]
_________________________________________________________________________________________________
conv2d_31 (Conv2D)           (None, 128, 128, 64)      73792     concatenate_6[0][0]
_________________________________________________________________________________________________
conv2d_32 (Conv2D)           (None, 128, 128, 64)      36928     conv2d_31[0][0]
_________________________________________________________________________________________________
conv2d_33 (Conv2D)           (None, 128, 128, 21)      1365      conv2d_32[0][0]
=================================================================================================
Total params: 7,706,325
Trainable params: 7,702,485
Non-trainable params: 3,840
_________________________________________________________________________________________________
```

And a small version looks like this:

```
Model: "model_2"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_3 (InputLayer)         [(None, 128, 128, 3)      0
_________________________________________________________________________________________________
conv2d_34 (Conv2D)           (None, 128, 128, 64)      1792        input_3[0][0]
_________________________________________________________________________________________________
batch_normalization_18 (BatchNo (None, 128, 128, 64)   256         conv2d_34[0][0]
_________________________________________________________________________________________________
conv2d_35 (Conv2D)           (None, 128, 128, 64)      36928       batch_normalization_18[0][0]
_________________________________________________________________________________________________
batch_normalization_19 (BatchNo (None, 128, 128, 64)   256         conv2d_35[0][0]
_________________________________________________________________________________________________
max_pooling2d_7 (MaxPooling2D)  (None, 64, 64, 64)     0           batch_normalization_19[0][0]
_________________________________________________________________________________________________
conv2d_36 (Conv2D)           (None, 64, 64, 128)       73856       max_pooling2d_7[0][0]
_________________________________________________________________________________________________
batch_normalization_20 (BatchNo (None, 64, 64, 128)    512         conv2d_36[0][0]
_________________________________________________________________________________________________
conv2d_37 (Conv2D)           (None, 64, 64, 128)       147584      batch_normalization_20[0][0]
_________________________________________________________________________________________________
batch_normalization_21 (BatchNo (None, 64, 64, 128)    512         conv2d_37[0][0]
_________________________________________________________________________________________________
max_pooling2d_8 (MaxPooling2D)  (None, 32, 32, 128)    0           batch_normalization_21[0][0]
_________________________________________________________________________________________________
conv2d_38 (Conv2D)           (None, 32, 32, 512)       590336      max_pooling2d_8[0][0]
_________________________________________________________________________________________________
batch_normalization_22 (BatchNo (None, 32, 32, 512)    2048        conv2d_38[0][0]
_________________________________________________________________________________________________
conv2d_39 (Conv2D)           (None, 32, 32, 512)       2359808     batch_normalization_22[0][0]
_________________________________________________________________________________________________
batch_normalization_23 (BatchNo (None, 32, 32, 512)    2048        conv2d_39[0][0]
_________________________________________________________________________________________________
conv2d_transpose_7 (Conv2DTrans (None, 64, 64, 128)    262272      batch_normalization_23[0][0]
_________________________________________________________________________________________________
concatenate_7 (Concatenate)  (None, 64, 64, 256)       0           conv2d_transpose_7[0][0]
                                                                   batch_normalization_21[0][0]
_________________________________________________________________________________________________
conv2d_40 (Conv2D)           (None, 64, 64, 128)       295040      concatenate_7[0][0]
_________________________________________________________________________________________________
conv2d_41 (Conv2D)           (None, 64, 64, 128)       147584      conv2d_40[0][0]
_________________________________________________________________________________________________
conv2d_transpose_8 (Conv2DTrans (None, 128, 128, 64)   32832       conv2d_41[0][0]
_________________________________________________________________________________________________
concatenate_8 (Concatenate)  (None, 128, 128, 128 0            conv2d_transpose_8[0][0]
                                                                   batch_normalization_19[0][0]
_________________________________________________________________________________________________
conv2d_42 (Conv2D)           (None, 128, 128, 64)      73792       concatenate_8[0][0]
_________________________________________________________________________________________________
conv2d_43 (Conv2D)           (None, 128, 128, 64)      36928       conv2d_42[0][0]
_________________________________________________________________________________________________
conv2d_44 (Conv2D)           (None, 128, 128, 21)      1365        conv2d_43[0][0]
=================================================================================================
Total params: 4,065,749
Trainable params: 4,062,933
Non-trainable params: 2,816
_________________________________________________________________________________________________
```

We see a vast comparison between parameters.

The performance of the models are measured across 3 independent trials, and the maximum metrics from each of the monitored metrics are plotted. We are considering a new Mean IoU metric where we do not account for the background of the image.



Overall, I think that this is rather inconclusive. Keep in mind that the mean iou metric does not account of the background so it is expected to find these so low with only 25 epochs…

The models all required roughly the same training time. This is likely due to the generator that is used to train them and that we compute the gradients on a GPU which is quite quick. Each of them takes ~28 seconds per epoch.

Original U-net	`800.1432586829997`
Medium U-net	`793.2822287939998`
Small U-net	`788.3507340299984`

In prediction, in 5 independent trials, the average times to predict all 1400 validation samples in was:

Original U-net	11.768321260999937
Medium U-net	9.85271608639996
Small U-net	9.281936669800052

According to the result above, we can see that using a small unet we can save more than 20% to predict the result but the performance of a small unet is not as good as the original unet. Medium unet can save only 17% of predicting time and keep the performance of the original unet. In conclusion, we can use medium unet as a more efficient and accurate model.

Conclusion:

Image semantic segmentation project is the process of partitioning a digital image into multiple segments, which is typically used to locate objects and boundaries in images. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics. In our project, we designed our model based on different research and compared their performance. We find the unet model has the best performance among our models. We also tune the optimizer, loss function and metrics. When we use SGD(lr=1e-4, decay=1e-3) with weighted loss function and mean_iou, it will give us best performance. At last, we reconstruct our unet model to make it have less parameters , less predicting cost and similar performance.

The current model we are using is easy to understand while the performance of these designs are not best for this project. For the further steps, we will try to work on the more complex models like deeplab v3, which has the best accuracy in the project leaderboard. Although we have tried to compare different optimizers, there are still other optimizer and learning rate schedules we didn't study on. In the future jobs, we will focus on more about the parameter choices.

Reference:

https://ilmonteux.github.io/2019/05/10/segmentation-metrics.html
https://towardsdatascience.com/metrics-to-evaluate-your-semantic-segmentation-model-6bcb99639aa2
https://github.com/ved27/MLIP-PROJECT/blob/master/PROJ_vgg16_ssd_train-voc2012-2007_pyramidal.ipynb
https://github.com/lidongyue12138/Image-Segmentation-by-Keras
https://www.kaggle.com/counter/image-segmentation-for-self-driving-cars
https://github.com/qubvel/segmentation_models
https://github.com/advaitsave/Multiclass-Semantic-Segmentation-CamVid/blob/master/Multiclass%20Semantic%20Segmentation%20using%20U-Net.ipynb
https://towardsdatascience.com/u-net-b229b32b4a71
https://towardsdatascience.com/review-fcn-semantic-segmentation-eb8c9b50d2d1
https://medium.com/datadriveninvestor/small-objects-detection-problem-c5b430996162
https://www.cs.toronto.edu/~tingwuwang/semantic_segmentation.pdf