Carvana Image Masking Challenge (Kaggle)

https://www.kaggle.com/c/carvana-image-masking-challenge

Project Background

This code challenge is hosted by Carvana (an online marketplace for buying and selling used cars). As part of their innovation, Carvana uses a custom rotating photo studio that captures 16 different projects of each vehicle in their inventory. They will like to use these images and superimpose them on a variety of backgrounds.

However, due to bright reflections and cars with similar colors as the background, this may lead to automation errors and thus, it requires a photo editor to manually change them.

Problem Statement

Manual editing of these image projections across thousands of individual cars is a time-consuming and expensive task. Hence, Carvana had approached Kaggle to host the online competition for users to develop an algorithm that generates masks for these images and classify each pixel in the mask and label them as 1 representing the car and 0 representing the background.

Dataset

In this competition, Kagglers were provided two datasets that were essential for training our algorithm and measuring the algorithm's performance on unseen data.

For the training data, we were given 5,088 training images that consists of 16 different projections for 318 vehicles. Also, 5,088 training masks were provided as our target output for our algorithm to learn from.

For the testing data, we have 100,064 testing images that consists of 16 different projections for 6,254 vehicles.

The resolution for each image and mask is 1,918 x 1,280 (width x height).

Machine learning algorithm and approach

This competition is a case of semantic image segmentation task where the U-Net (CNN network) can be employed to classify each pixel in the input image as 1 representing car and 0 representing background.

Training a U-Net network allows it to learn contextual information in its encoding path (i.e. contracting) and localize information in its decoding path (i.e. expanding). Due to its symmetric layering and large number of feature maps used within the network, U-Net is able to transfer feature information effectively from an input image to output probabilities computed by activation function in its final layer. As such, U-Net is used for this project.

Two U-Net networks will be trained – where input training images were resized to the format of (height, width, channels) in 128 x 128 x 3 (first network) and 256 x 256 x 3 (second network).

The architecture of U-Net works as follows:

<u>Global Parameter Settings</u>

- Apart from initial number of filters, center number of filters number of classes and maximum number of epochs, the remaining hyperparameters were tuned by assessing the network's loss function (BCE_DICE_LOSS – which will be discussed later). Hyperparameters resulting in lower loss are preferred.

	U-Net 128 x 128 x 3	**U-Net 256 x 256 x 3**
Initial no of filters	64	32
Center no of filters	1024	1024
No of classes (output mask)	1	1
Learning rate	0.0001	0.0001
Max no of epochs	30	30
Batch size	4	4
Threshold (output probabilities)	0.5	0.5

<u>Encoding Path</u>

- In encoding (contracting) path of U-Net network, it consists of 4 blocks for U-Net 128 and 5 blocks for U-Net 256
- Each block is constructed in the order of:
  - 3 x 3 convolutional layer => batch normalization => activation ReLU.
  - 3 x 3 convolutional layer => batch normalization => activation ReLU.
  - 2 x max pooling layer.
- We note that the number of feature maps doubles after each pooling (see below for the doubling sequence):
  - U-Net 128: 64 => 128 => 256 => 512.
  - U-Net 256: 32 => 64 => 128 => 256 => 512.

<u>Center</u>

- This refers to the center portion between the encoding and decoding paths.
- It is constructed in the order of:
  - 3 x 3 convolutional layer => batch normalization => activation ReLU.
  - 3 x 3 convolutional layer => batch normalization => activation ReLU.
- We note that in center portion, 1,024 filters are used regardless of the size of input image and in this, U-Net 128 or U-Net 256.

<u>Decoding Path</u>

- In decoding (expanding) path of U-Net network, it also consists of 4 blocks for U-Net 128 and 5 blocks for U-Net 256
- Each block is constructed in the order of:
  - 2 x 2 deconvolutional layer (for upsampling) with stride 2.
  - Concatenation with the corresponding cropped feature map from the enconding path (for example in the case of U-Net 128, we concatenate block 5 from decoding path with block 4 from encoding path. Note that the center portion is not considered as a block of layers. In the case of U-Net 256, we concatenate block 6 from decoding path with block 5 from encoding path.)
  - 3 x 3 convolutional layer => batch normalization => activation ReLU.

Final layer (output layer)

- The output from final block in decoding path is passed through a 1 x 1 convolutional layer with 1 filter where filter represents the number of classes we are trying to classify.
- By using the weights and bias term initialized and forward passed in the network, we obtain output probabilities by passing the weights and bias term through the sigmoid activation function in our final layer.
- Subsequently, gradient is calculated by using our selected optimizer (i.e. RMSprop) on the error between predicted mask and ground truth mask multiplied by our sigmoid derivative on our predictions and then taking the dot product with our features.
- This gradient value is then used to update our weights and bias term in backward pass so as to ensure that our loss is moving lower gradually as network training continues to proceed.

Train / validation data split

- Our 5,088 training images were split by 80 / 20 ratio into training and validating data.
- Validation data will be used to evaluate the loss and any model metrics at the end of each epoch. The network will not be trained on this data.

Callbacks

- The callbacks used in our network training were:
  - EarlyStopping: stop training when our BCE_DICE_LOSS stopped improving / decreasing after 8 number of epochs (patience: 8). The minimum change in loss to qualify as improvement is 1e-4.
  - ReduceLROnPlateau: reduce learning rate when our BCE_DICE_LOSS stopped improving / decreasing after 4 number of epochs (patience: 4). The factor by which to reduce learning rate is calculated as: new_learning_rate = factor * previous_learning_rate. Factor has been set as 0.1 in our case.
  - ModelCheckPoint: to save our network weights for training iteration that returned the best BCE_DICE_LOSS.
  - TensorBoard: to create training logs whenever network is being trained.

Model Compilation

The following configurations were used for our network:

- Optimizer: RMSprop
  - Is selected over Stochastic Gradient Descent as preliminary modelling showed that RMSprop returned lower loss.
- Loss: BCE_DICE_LOSS
  - Is calculated by: bce_dice_loss = binary_crossentropy + dice_loss (where dice_loss = 1 – dice_coefficient)
- Metrics: DICE_COEFF
  - Measures pixel-wise agreement between predicted mask and its corresponding ground truth mask.

Training and validating generators

- These generators were created to extract batches of training / validating images with their corresponding masks during training and validating process.
- For training generator, input images were augmented in three different ways:
  1. Changing input image's hue, saturation and value.
  2. Changing input image's and mask's size and position by scaling, rotating and translating.
  3. Changing input image's and mask's position by flipping them horizontally.
- The rationale for augmenting them is to reduce overfitting by our network and improve its generalizability over unseen (or test) dataset.

Model training

- .fit_generator() was called on our network to start training process.

Output classification

- With predictions obtained by calling .predict_on_batch() on our trained network, we obtained output probabilities.
- As in this competition, Kaggle only accepts run-length encoding on the test masks' pixel values, a run length encoder function was written to convert our binary labels to pairs of values that contain a start position and a run length.

Model Evaluation

- For the submission of test masks from employing U-Net 128 and U-Net 256, we have the following scores from Kaggle:

	U-Net 128 x 128 x 3	**U-Net 256 x 256 x 3**
Private Score	0.990301	0.994220
Public Score	0.990692	0.994284

Results, findings and issues encountered

- Some of the lessons learned were:
  1. To devote more time to visualizations.
  - There was a training mask where a hole was discovered. This meant that classification accuracy and training loss from our network will be negatively impacted.

  2. Develop strong understanding of input / output throughout the whole project pipeline.
  - Initial implementation of U-Net network was done in TensorFlow, however, during code testing the returned results was buggy. Hence, Keras (with TensorFlow backend) was used instead.

- Firm grasp of spatial visualization is important as well because it will aid in our understanding on how data is manipulated and transformed as we move from one part of project pipeline to another.

3. Explore other types of neural network architectures.

- Top solutions utilized ensemble of networks to improve their scores further from just using U-Net network.

Relevant references or information

- Keras documentations.
- Kaggle kernels.
- Stackoverflow discussions.
- A lot of google searches on research papers and articles written on the subject of semantic image segmentation and U-Net network.