Semantic Segmentation on a Driving Dataset

Langchun Zhou

Summary

The problem is as follows: given a picture of traffic, categorize each pixel of that picture as belonging to exactly one of multiple categories of traffic elements. The motivation behind this task was to explore a deep learning technology that will help to reduce the incidence and deadliness of traffic accidents through implementation in self-driving cars. This task is an instance of semantic segmentation, which is the act of categorizing each pixel of an image as belonging to exactly one of multiple categories of elements that compose the image. In this case, semantic segmentation was performed on a preprocessed sample of the Mapillary Vistas Dataset, first with an autoencoder and then with a U-Net which addressed some of the problems of the autoencoder with regard to the task.

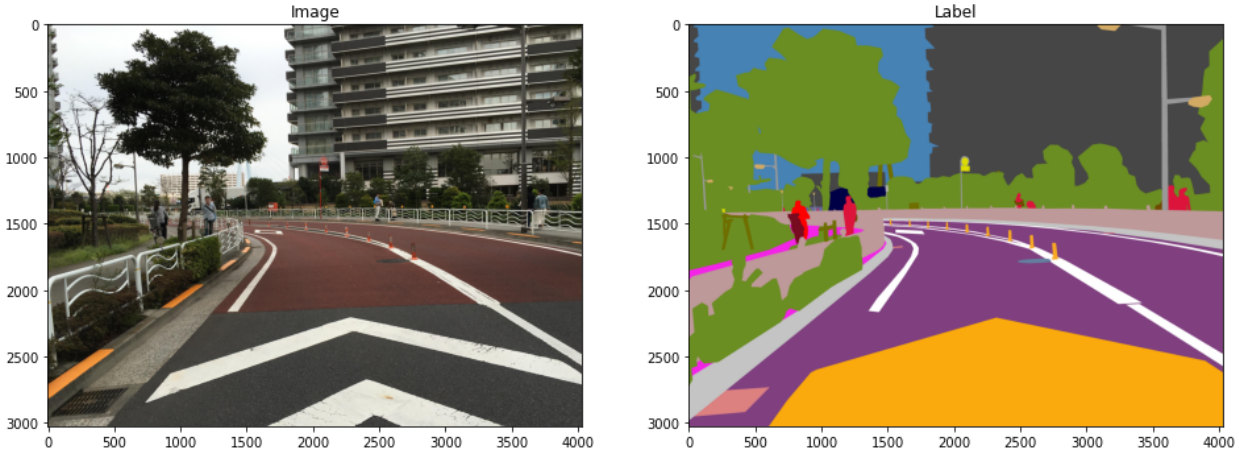
Installation

The notebook runs on a standard Anaconda Tensorflow installation for Windows, as per https://www.anaconda.com/products/distribution and https://www.tensorflow.org/install/pip. Download and extract the notebook and its dependencies, then download the Mapillary Vistas Dataset from https://www.mapillary.com/dataset/vistas and extract it to the root directory of the notebook. Name the dataset "data" and run the notebook in the Tensorflow environment.

Dataset

Visualization

In the Mapillary Vistas Dataset, inputs are JPGs of traffic and labels are PNGs. For each label, each pixel is categorized by a color that maps to the traffic element that pixel constitutes:



For this study, v2.0 labelings were used, for which the mappings of colors to categories were stored in data/config_v2.0.json:

```
config_file = open('data/config_v2.0.json')
config = json.load(config_file)
```

config_v2.0.json describes categories such as “nature--vegetation”, “nature--sky”, and “construction--flat--road”, which map to RGB colors such as [107, 142, 35], [70, 130, 180], and [128, 64, 128].

Preprocessing

Preprocessing was motivated by the training platform: a Dell XPS 7590 laptop with an Nvidia GeForce GTX 1650 GPU and Intel i7-9750H CPU. An important objective was to make the models run on computers of equivalent power in order to serve as toy examples. The first step toward this objective was to sample smaller training, validation, and test datasets from the full Mapillary Vistas Dataset in order to reduce training times:

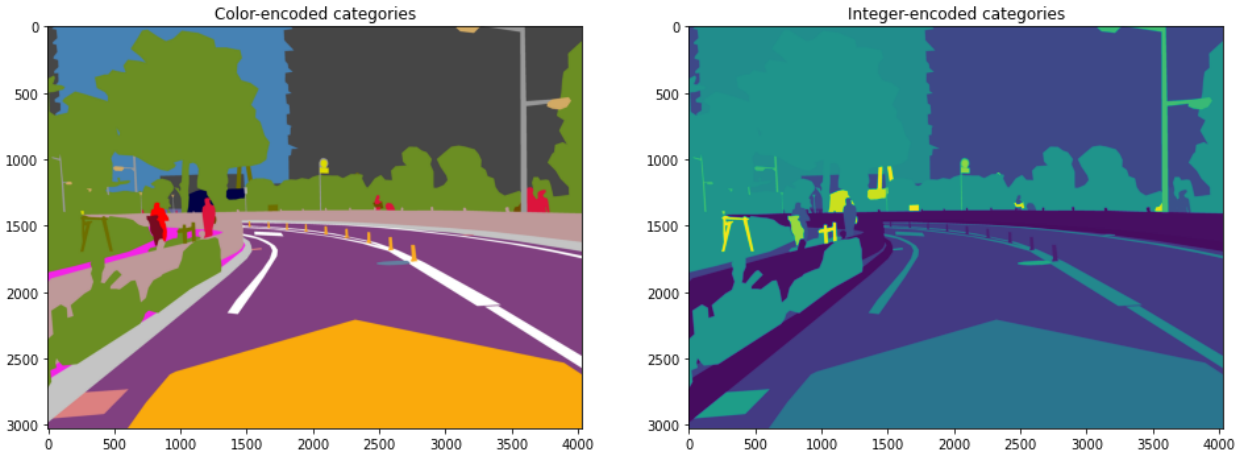
```
if preprocess == True:
    file_names = random.sample(os.listdir(train_x_src_dir), train_size)
    process_images(train_x_src_dir, train_x_dir, file_names)

    file_names = list(map(lambda x: set_ext(x, '.png'), file_names))
    process_labels(train_y_src_dir, train_y_dir, file_names)

    file_names = random.sample(os.listdir(val_x_src_dir), val_size)
    process_images(val_x_src_dir, val_x_dir, file_names)

    file_names = list(map(lambda x: set_ext(x, '.png'), file_names))
    process_labels(val_y_src_dir, val_y_dir, file_names)

    file_names = random.sample(os.listdir(test_src_dir), test_size)
    process_images(test_src_dir, test_dir, file_names)
```

The samples consisted of 800 training images and labels, 100 validation images and labels, and 100 test images, all stored in data_small. These images and labels were also downsized to 512px ✕ 512px, and the labels were converted from color encodings of categories to integer encodings of categories in order to save space and provide a useful format for training Keras models:

```
categories_to_colors = list(map(lambda x: x['color'], config['labels']))
num_categories = len(categories_to_colors)
```

```
def to_categories(colors):
    categories = np.zeros((colors.shape[0], colors.shape[1], 1))

    for category in range(num_categories):
        color = categories_to_colors[category]
        mask = np.all(colors == color, axis=2)
        categories[mask] = [category]

    return categories
```

Each category is implicitly stored as an index, indexing to the color that represents it in `categories_to_colors`. `to_categories(colors)` returns an integer-encoded label from a color-encoded one: it initializes an output array with the same width and height of the input, then for each category it masks the areas of the output where the input's color equals that of the category, and sets those masked areas of the output to the index of the category. The resulting integer-encoded output is saved as a grayscale PNG, visualized below in Viridis:



By the same pattern, integer-encoded labels can be converted back to color encoding for easier visualization:

```
def to_colors(categories):
    colors = np.zeros((categories.shape[0], categories.shape[1], 3))

    for category in range(num_categories):
        mask = np.all(categories == category, axis=-1)
```

```
        colors[mask] = categories_to_colors[category]
    return colors
```

These functions were modified from S. Humbarwadi's `color_to_labels(color_img)` function in his U-Net example on GitHub (Humbarwadi, 2018, cell 4).

Autoencoder

Recall that our goal is to predict a 512px ✕ 512px semantically segmented label for a 512px ✕ 512px image. According to Jordan (2018), one naive way to do this would be to stack convolutional layers with "same" padding in order to preserve resolution and transform the input into a prediction. This is too expensive though, so encoder/decoder structures are often used instead.

Implementation

A naive autoencoder was implemented in order to get a baseline performance. It follows the pattern of the convolutional autoencoder from Harvard Summer School CSCI S-89 Summer 2022 Lecture 9 (Kurochkin, 2022):

```
num_features_0 = 4
input_shape = (input_width, input_height, 3)

encoder = models.Sequential()
encoder.add(layers.Conv2D(num_features_0, kernel_size=3, padding='same',
    activation='relu', input_shape=(input_shape)))
encoder.add(layers.MaxPool2D(pool_size=2))
encoder.add(layers.Conv2D(num_features_0 * 2, kernel_size=3, padding='same',
    activation='relu'))
encoder.add(layers.MaxPool2D(pool_size=2))
encoder.add(layers.Conv2D(num_features_0 * 4, kernel_size=3, padding='same',
    activation='relu'))
encoder.add(layers.MaxPool2D(pool_size=2))
```

```
decoder = models.Sequential()

decoder.add(layers.Conv2DTranspose(
    num_features_0 * 2,
    kernel_size=3,
    strides=2,
    padding='same',
    activation='relu',
    input_shape=(input_width // 8, input_height // 8, num_features_0 * 4)))

decoder.add(layers.Conv2DTranspose(num_features_0, kernel_size=3, strides=2,
    padding='same', activation='relu'))
decoder.add(layers.Conv2DTranspose(num_categories, kernel_size=3, strides=2,
    padding='same', activation='softmax'))

autoencoder = models.Sequential([encoder, decoder])
autoencoder.compile(loss='sparse_categorical_crossentropy',
optimizer=optimizers.Adam())
```

The low feature depth is a consequence of the project being a toy model that needs to respect computation and memory constraints. The last layer uses a softmax activation, producing for each pixel a vector of 124 probabilities (one probability per category). This means that the last layer alone produces $512 \times 512 \times 124$ elements, which further motivates the low batch size of 5 in order to stay within memory constraints.

Recall that the labels are integer-encoded. This allows us to use sparse_categorical_crossentropy as a loss function on the vectors produced by the softmax activation layer. If the labels were instead one-hot encoded, they would take up 124 times more space and we'd be using categorical_crossentropy as a loss function instead.

Training

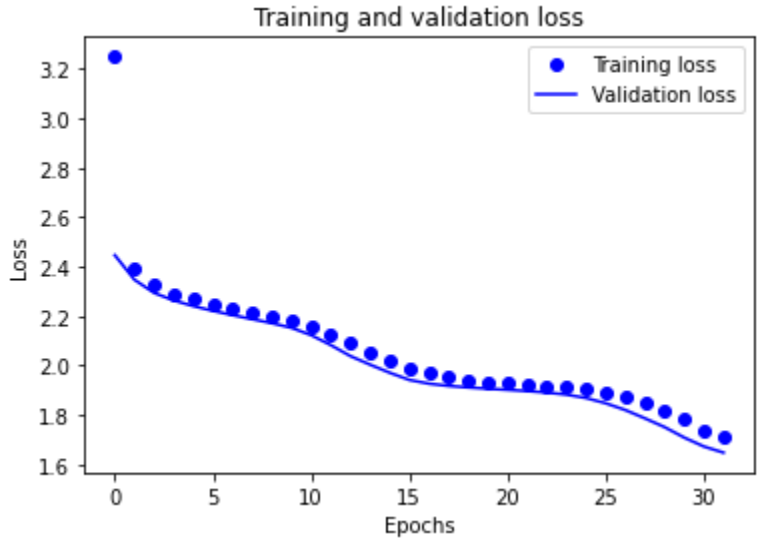
```
history = model.fit(
    train_gen,
    batch_size=batch_size,
    steps_per_epoch=train_size//batch_size,
```

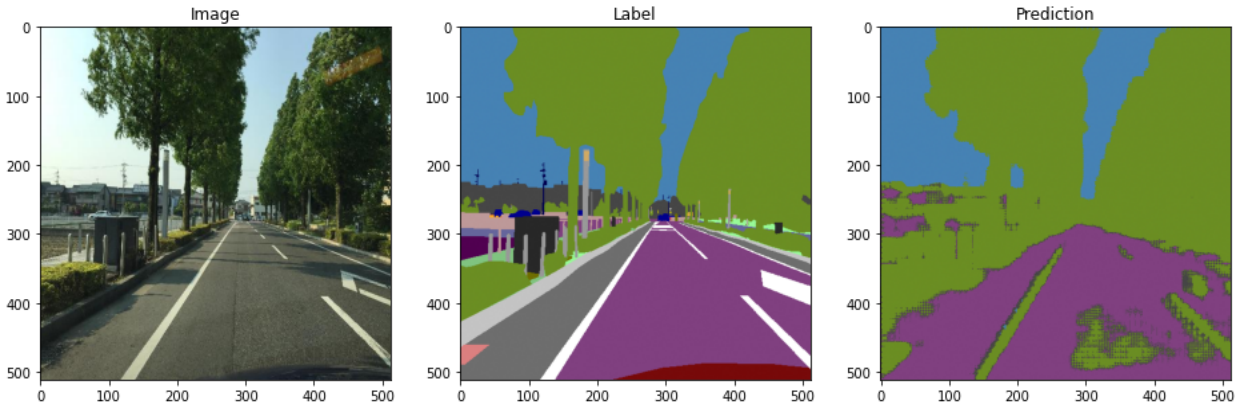
```
        epochs=num_epochs,
        validation_data=val_gen,
        validation_steps=val_size//batch_size
    )

    model = models.Sequential([model, Argmax()])
```

After training, a custom Argmax layer is appended to the softmax activation layer in order to get the most probable category per pixel, thereby reducing the prediction from a size of (512, 512, 124) to (512, 512) and recovering an integer encoding:

```
class Argmax(layers.Layer):
    def __init__(self):
        super(Argmax, self).__init__()

    def call(self, inputs):
        return tf.math.argmax(inputs, axis=-1)
```

This code was referenced from Jan Jongboom's question and answers on StackOverflow on how to make a custom Argmax layer (Jongboom, 2019). After training the entire model and obtaining integer-encoded outputs through Argmax, the results were quite poor:

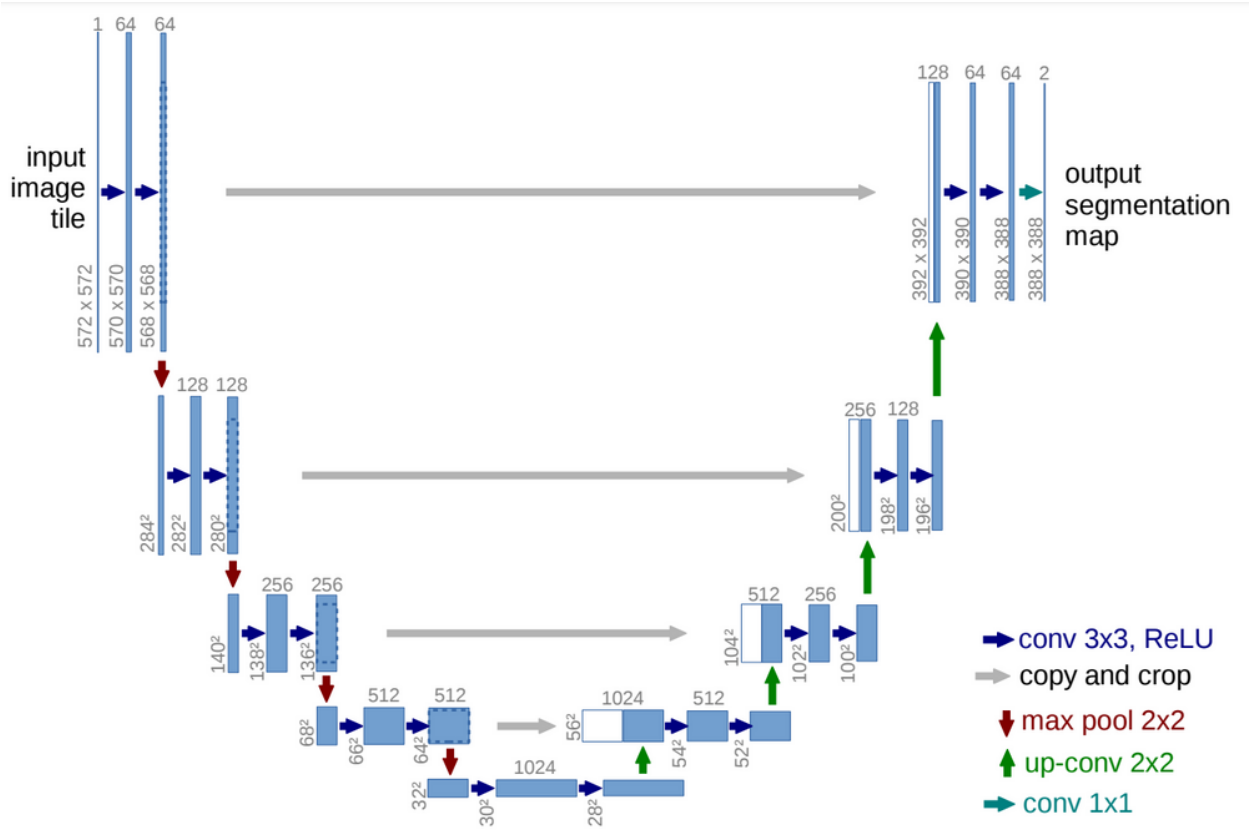




The model only recognized vegetation, skies, and roads, and contained artifacts from upsampling: too much information was lost through downsampling. In order to preserve information and resolution, the next model used was U-Net.

U-Net

U-Net is an encoder/decoder network with skips from encoders to decoders in order to retain information before downsampling. The network can be visualized as blocks: for each block, its height is its resolution and its width is its feature depth. Input progresses “right” through blocks, going “down” when downsampled and “up” when upsampled:



Ronneberger et al., 2015, p.235

This visualization produces the U shape which gives the model its name. Notice the skips from encodings to decoders of equal resolution: this retains the information from those encodings before it is downsampled and passed to the next encoder.

Implementation

Implementation followed Bharath K.'s functional style in his PaperspaceBlog post (Bharath K., 2021). The implementation was chosen for its clarity and simplicity:

```
def convolve(my_input, num_features):
    conv_1 = layers.Conv2D(num_features, kernel_size=3, activation='relu',
        padding='same')(my_input)
    conv_2 = layers.Conv2D(num_features, kernel_size=3, activation='relu',
        padding='same')(conv_1)
    return conv_2
```

```
def encode(my_input, num_features):
    encoding = convolve(my_input, num_features)
    pool = layers.MaxPool2D(2)(encoding)
    return encoding, pool

def decode(my_input, skip, num_features):
    upsample = layers.Conv2DTranspose(num_features, kernel_size=2, strides=2,
        padding='same')(my_input)
    concat = layers.Concatenate()([upsample, skip])
    out = convolve(concat, num_features)
    return out

def get_unet():
    my_input = keras.Input(input_shape)

    skip_1, encoding_1 = encode(my_input, 4)
    skip_2, encoding_2 = encode(encoding_1, 8)
    skip_3, encoding_3 = encode(encoding_2, 16)
    skip_4, encoding_4 = encode(encoding_3, 32)

    convolution = convolve(encoding_4, 64)

    decoding_1 = decode(convolution, skip_4, 32)
    decoding_2 = decode(decoding_1, skip_3, 16)
    decoding_3 = decode(decoding_2, skip_2, 8)
    decoding_4 = decode(decoding_3, skip_1, 4)

    out = layers.Conv2D(num_categories, kernel_size=1, padding='same',
        activation='softmax')(decoding_4)
    model = keras.Model(my_input, out)
    return model

unet = get_unet()
unet.compile(loss='sparse_categorical_crossentropy',
optimizer=optimizers.Adam())
```

As prescribed in Ronneberger et al.'s original paper (2015), each convolutional encoder block returns an encoding before downsampling, and a downsampling. The encoder is passed to a decoder as skip, and the downsampling is passed to the next encoder. In turn, each convolutional decoder block takes a skip and concatenates it with its upsample from the previous coding, representing a recovery of information from the "other side of the U". As with the

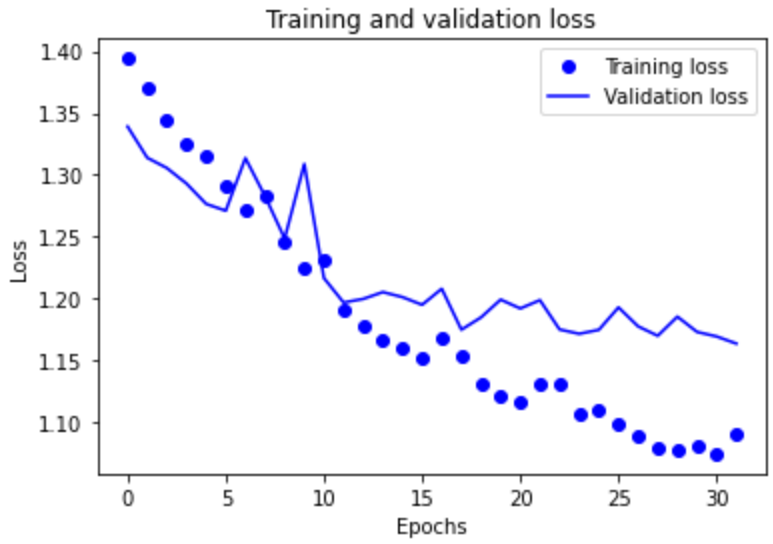
autoencoder, this toy model has vastly reduced feature depth (from an initial 64 features in the original paper to a mere 4 here) in keeping with the memory constraints of a toy example.

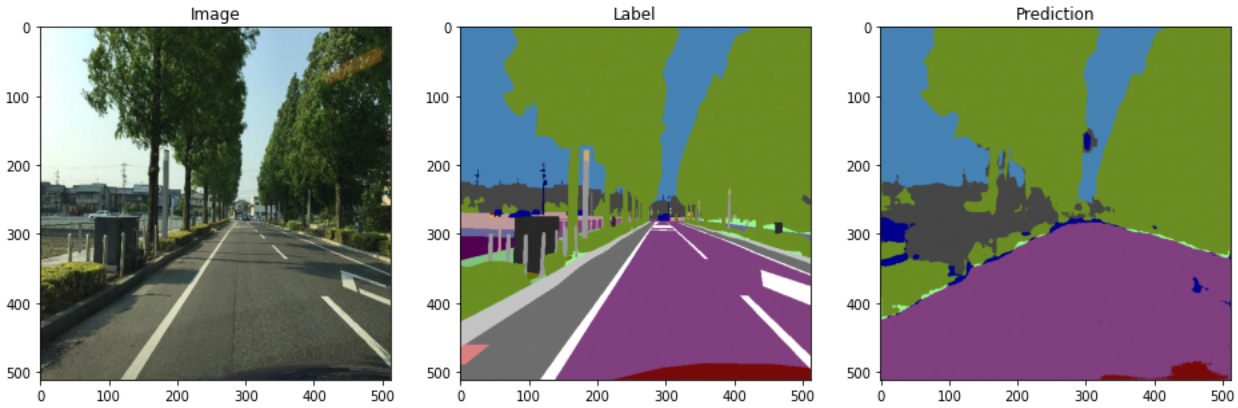
Training

To retain comparability, the U-Net was trained for the same number of epochs with the same loss function and optimizer as those of the autoencoder:

```
load_autoencoder = True
save_autoencoder = False
autoencoder_dir = 'autoencoder'
num_epochs = 32

...

autoencoder = get_trained_model(
    autoencoder,
    load_autoencoder,
    save_autoencoder,
    num_epochs,
    autoencoder_dir
)

...

load_unet = True
save_unet = False
unet_dir = 'unet'
num_epochs = 32

unet = get_trained_model(
    unet,
    load_unet,
    save_unet,
    num_epochs,
    unet_dir
)
```

Results were much better for the U-Net:





The U-Net's initial loss was lower than the naive autoencoder's final loss. In addition to trees, foliage, and road, the U-Net also began to recognize the videographer's car, other cars, and background elements. Artifacts were removed thanks to information restored through the skip connections.

More miscategorizations also occurred. The U-Net mistook a shadow and lane marking in in the right lane for a car, as it also did for the parking lot on the left. Elements of the midground were also mistaken for elements of the background. This shows that, with more potential for fine-grained correctness, the model also has more potential for dangerous misinterpretation.

Finally, the model began to overfit around the 11th epoch, as indicated by the divergence between training and validation loss. This was to be expected as the model had over an order of magnitude more trainable parameters than that of the autoencoder (122,340 vs. 7,616) while training on a small toy dataset. More sophisticated, industrial-quality models have used techniques such as dropouts and batch normalization to reduce overfitting and achieve state-of-the-art semantic segmentation results.

Conclusion

Semantic segmentation is an important early step in processing driving information. A natural progression of this problem is instance segmentation, which is the problem of identifying individual instances of the same category of traffic element. Both of these problems are generalizable to the third dimension with lidar data, ensuring more fidelity to the actual traffic environment. In addition, adding the element of time to these problems, i.e. solving multiple segmentation problems over time, introduces the potential for motion prediction and planning in self-driving strategies. In this way, semantic segmentation is an early step toward safer automated traffic for all: drivers, passengers, and pedestrians.

YouTube Link

https://youtu.be/bUL0zC7l4xc

References

Humbarwadi, S. (2018, 15 October). *Keras-Unet / data.ipynb*. GitHub. Retrieved August 5, 2022 from https://github.com/srihari-humbarwadi/Keras-Unet/blob/master/data.ipynb.

Jongboom, J. (2019, 21 June). *Keras output single value through argmax*. StackOverflow. Retrieved August 5, 2022 from https://stackoverflow.com/questions/56704669/keras-output-single-value-through-argmax

Jordan, J. (2018, 21 May). *An overview of semantic image segmentation*. Jeremy Jordan. Retrieved August 5, 2022 from https://www.jeremyjordan.me/semantic-segmentation/

K., Bharath. (2021). *U-Net Architecture For Image Segmentation*. PaperspaceBlog. Retrieved August 5, 2022 from https://blog.paperspace.com/unet-architecture-image-segmentation/.

Kurochkin, Dmitry. *Summer 2022 Lecture 9*. Harvard Canvas. Retrieved August 5, 2022 from https://canvas.harvard.edu/.

Neuhold, G., Ollmann, T., Rota Buló, S., & Kontscheider, P. (2017). *The Mapillary Vistas Dataset for Semantic Understanding of Street Scenes*. Mapillary. Retrieved August 5, 2022 from https://www.mapillary.com/dataset/vistas.

Ronneberger, O., Fischer, P., Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. In: Navab, N., Hornegger, J., Wells, W., Frangi, A. (eds) *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015. MICCAI 2015. Lecture Notes in Computer Science()*, vol 9351. Springer, Cham. Retrieved August 5, 2022 from https://doi.org/10.1007/978-3-319-24574-4_28