Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Segmentación Semántica

Informe de Tarea 6

Alumno: José Rubio
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla
Ayudantes: Juan Pablo Cáceres B.
Javier Smith D.
Hans Starke D.
José Villagrán E.

Fecha de realización: 6 de enero de 2021
Fecha de entrega: 6 de enero de 2021
Santiago, Chile

Índice de Contenidos

Índice de Figuras

Índice de Códigos

1. Introducción.

La visión artificial es una disciplina científica que que afronta varios problemas generales, uno de ellos es la división de una imagen digital en varios sectores, con la finalidad de separar la imagen en una cantidad de regiones deseadas.

La segmentación semántica corresponde al reconocimiento de regiones de una imagen donde a cada pixel se le asigna una clase, por lo que en las figuras segmentadas se aprecian se aprecian diferentes tipos de objetos, este se desarrolla mediante una clasificación pixel a pixel. La finalidad de la segmentación es localizar regiones con significado, donde esta es útil para localizar objetos como para detectar los bordes de una imagen. El resultado de la segmentación aplicada corresponde a una imagen corresponde a una etiqueta atribuida a cada pixel o puede ser representada como un conjunto de contornos, por lo que debido a la cantidad de clasificaciones entonces es razonable el uso de *deep learning* para el desarrollo de esta actividad.

Este trabajo es sumamente interesante debido a las aplicaciones que se le pueden dar, como el reconocimiento necesario para procesar la información de un vehículo autónomo, donde se tiene un enfoque más preciso sobre las detecciones realizadas debido al reconocimiento de las clases pixel a pixel. Debido a esto, la base de datos utilizada en esta experiencia corresponde al `Dataset Kitti`, el cual contiene varios archivos comprimidos que poseen diversas figuras captadas por un vehículo en movimiento, donde también se tienen las **máscaras de segmentación** de las imágenes, cabe destacar que este trabajo es bastante complejo ya que requiere analizar cada pixel de las imágenes para atribuirlos a una etiqueta.

En este informe se presentarán modelos que desarrollan una segmentación semántica en la base de datos mencionada anteriormente, donde estos sistemas corresponden a una red UNet salvo que los valores de sus parámetros fueron tomados en diferentes puntos durante su entrenamiento. Esto se desarrolló con la finalidad de comparar el rendimiento del sistema durante su entrenamiento y después de finalizarlo, donde los resultados fueron evaluados mediante el análisis visual de las máscaras obtenidas y comparándolas con las máscaras reales, donde además se analiza el desempeño del entrenamiento visualizando las pérdidas gráficamente.

2. Desarrollo.

Para el proceso de la segmentación se realizó el siguiente desarrollo: se creó un objeto para cargar los datos del dataset Kitti, y luego usando funciones predefinidas extraídas del siguiente link[1] se realizó el entrenamiento y la evaluación del modelo. Sin embargo las máscaras predichas fueron obtenidas mediante una función personalizada. El desarrollo específico de esto se pueden ver en las siguientes secciones.

2.1. Dataset *Kitti.*

Este dataset contiene una gran cantidad de imágenes de un vehículo en movimiento, el cual pasa por diversas calles y caminos. Para la obtención y extracción de este dataset se usó el siguiente bloque:

```
!wget https://s3.eu-central-1.amazonaws.com/avg-kitti/data_semantics.zip
!unzip 'data_semantics.zip'
```

Código 1: Descarga y extracción del zip Kitti

Estas líneas fueron ejecutadas en diferentes bloques, donde la primera obtuvo los datos del url especificado y guardó el zip en la interfaz de colab, mientras que la segunda línea extrajo el archivo comprimido en 2 carpetas: una con las imágenes del conjunto de entrenamiento y otra con las imágenes para evaluar la red.

Dada la naturaleza del problema, la carpeta `training` contuvo imágenes y máscaras, donde estas últimas vinieron en una carpeta llamada `semantic`. Los valores de esta máscara correspondieron a las etiquetas, sin embargo las máscaras contuvieron 31 valores posibles, por lo que se usó la función `kitty_inverse_map_1channel()` para llevarlas a 12 clases (Esta función es mostrada en el anexo). Este último proceso se aplicó en la construcción del objeto `KittiDataset`.

2.2. Clase KittiDataset.

Se solicitó la construcción de este dataset, para poder implementar los datasets en `pytorch`. Para ello se utilizó el siguiente bloque:

```
class KittiDataset(Dataset):
    def __init__(self, imgs_dir, masks_dir, read_mask, scale=1, mask_suffix=''):
        super(KittiDataset, self).__init__()
        self.imgs_dir = imgs_dir
        self.masks_dir = masks_dir
        self.read_mask = read_mask
        self.scale = scale
        self.mask_suffix = mask_suffix
        assert 0 < scale <= 1, 'Scale must be between 0 and 1'

        self.ids = [splitext(file)[0] for file in listdir(imgs_dir)
                    if not file.startswith('.')]
        logging.info(f'Creating dataset with {len(self.ids)} examples')
```

[1] https://github.com/milesial/Pytorch-UNet

```
14
15  def __len__(self):
16      return len(self.ids)
17
18  @classmethod
19  def preprocess(cls, pil_img, scale):
20      w, h = pil_img.size
21      newW, newH = int(scale * w), int(scale * h)
22      assert newW > 0 and newH > 0, 'Scale is too small'
23      pil_img = pil_img.resize((newW, newH))
24
25      img_nd = np.array(pil_img)
26
27      if len(img_nd.shape) == 2:
28          img_nd = np.expand_dims(img_nd, axis=2)
29
30      # HWC to CHW
31      img_trans = img_nd.transpose((2, 0, 1))
32      if img_trans.max() > 1:
33          img_trans = img_trans / 255
34
35      return img_trans
36
37  def __getitem__(self, i):
38      idx = self.ids[i]
39      img_file = glob(self.imgs_dir + idx + '.*')
40      img = Image.open(img_file[0])
41      if self.read_mask:
42        mask_file = glob(self.masks_dir + idx + self.mask_suffix + '.*')
43        assert len(mask_file) == 1, \
44            f'Either no mask or multiple masks found for the ID {idx}: {mask_file}'
45        assert len(img_file) == 1, \
46            f'Either no image or multiple images found for the ID {idx}: {img_file}'
47        mask = Image.open(mask_file[0])
48
49        assert img.size == mask.size, \
50            f'Image and mask {idx} should be the same size, but are {img.size} and {mask.size}'
51
52        mask = kitty_inverse_map_1channel(np.array(mask, dtype=np.int32))
53      else:
54        mask_empty = np.array([])
55      img = self.preprocess(img, self.scale)
56      if self.read_mask:
57        return {
58            'image': torch.from_numpy(img).type(torch.FloatTensor),
59            'mask': torch.from_numpy(mask).type(torch.FloatTensor)}
60      else:
61        return {
62            'image': torch.from_numpy(img).type(torch.FloatTensor),
63            'mask': mask_empty}
```

Código 2: Clase KittiDataset

El código base de este objeto fue extraido del siguiente link[2], donde se realizaron modificaciones partiendo desde el nombre del objeto hasta las cancelaciones a algunas llamadas en la función **___getitem___**, un cambio destacado fue la incorporación del parámetro **read_mask** en la inicialización, con lo que varias instancias **assert** fueron canceladas al momento de desarrollar esta función, esto se desarrolló con el fin de no cargar máscaras, ya que el conjunto de prueba no contuvo máscaras, solamente el conjunto de entrenamiento.

La función más destacada fue **___getitem___** fue la más destacada, cuya salida fue un diccionario que contuvo la imagen del item seleccionado junto con su máscara si la variable **read_mask** fuese **True**, en caso de que esta variable fuese **False**, entonces el tensor de la máscara obtenido fue un tensor vacío.

2.3. Red UNet.

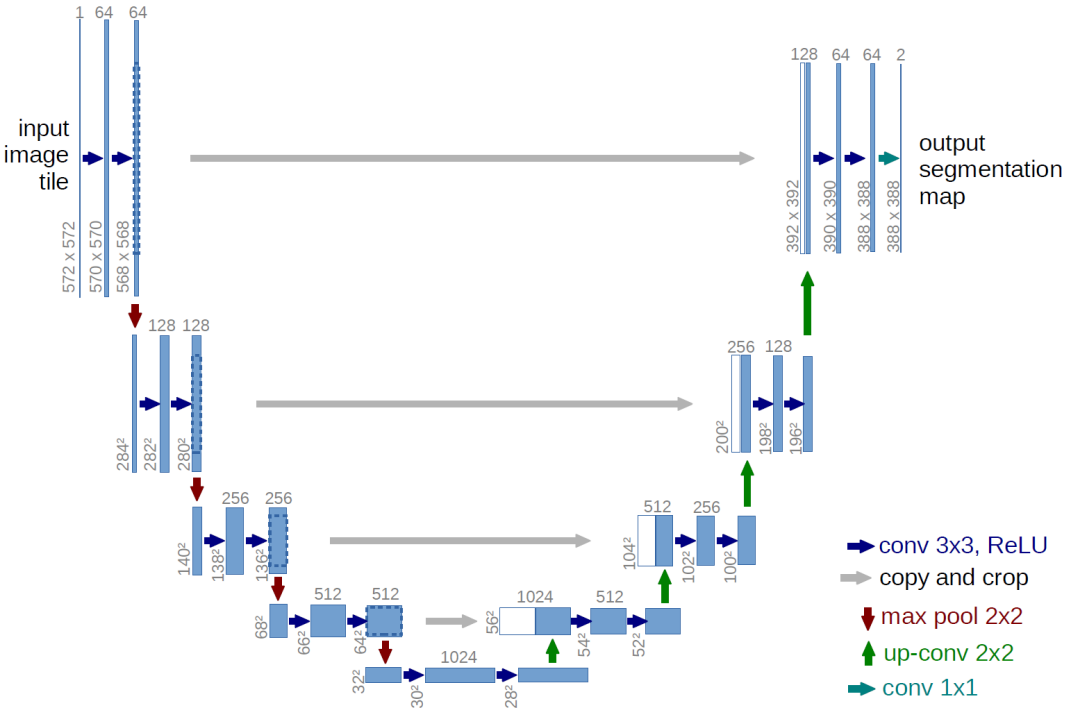

Figura 1: Arquitectura de la UNet, los números en la parte superior de cada bloque corresponde a la cantidad de canales, mientras que los que se ubican en el lateral corresponden a las dimensiones de la imagen procesada.

Esta red fue obtenida desde el repositorio de **github** entregado, para ello se usaron los siguientes comandos.

```
if not os.path.exists('unet_parts.py'):
  !wget https://raw.githubusercontent.com/milesial/Pytorch-UNet/master/unet/unet_parts.py

```

[2] https://raw.githubusercontent.com/milesial/Pytorch-UNet/master/utils/dataset.py

```
if not os.path.exists('unet_model.py'):
  !wget https://raw.githubusercontent.com/JossRubio/Pytorch-UNet/patch-1/unet/unet_model.py

if not os.path.exists('eval.py'):
  !wget https://raw.githubusercontent.com/milesial/Pytorch-UNet/master/eval.py

if not os.path.exists('dice_loss.py'):
  !wget https://raw.githubusercontent.com/milesial/Pytorch-UNet/master/dice_loss.py
```

Código 3: Comando para cargar la red UNet

Cargando estas librerías, entonces se pudo llamar al objeto `UNet` y `eval_net` de la siguiente forma:

```
from unet_model import UNet    # Linea para cargar Unet
from eval import eval_net      # Importación de eval_net
```

Código 4: Carga de la red UNet y la función de validación

El código del objeto que contiene esta red se encuentra en este link[3]. De todas formas también se encuentra en el anexo la programación de esta red.

2.4. Entrenamiento de la red.

El entrenamiento fue el proceso más complejo, para ello se adaptó el archivo `train.py` del github. El código adaptado completo fue el siguiente:

```
def train_net(net, device, epochs=7, batch_size=1, lr=0.001, val_percent=0.1,
              save_cp=True, img_scale=1):

    dataset = KittiDataset(imgs_dir=dir_img, masks_dir=dir_mask, read_mask=True, scale=
     ↪ img_scale)
    n_val = int(len(dataset) * val_percent)
    n_train = len(dataset) - n_val
    train, val = random_split(dataset, [n_train, n_val])
    train_loader = DataLoader(train, batch_size=batch_size, shuffle=True, num_workers=8,
     ↪ pin_memory=True)
    val_loader = DataLoader(val, batch_size=batch_size, shuffle=False, num_workers=8,
     ↪ pin_memory=True, drop_last=True)

    writer = SummaryWriter(comment=f'LR_{lr}_BS_{batch_size}_SCALE_{img_scale}')
    global_step = 0

    optimizer = optim.RMSprop(net.parameters(), lr=lr, weight_decay=1e-8, momentum=0.9)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min' if net.n_classes > 1 else '
     ↪ max', patience=2)
    if net.n_classes > 1:
        criterion = nn.CrossEntropyLoss()
    else:
        criterion = nn.BCEWithLogitsLoss()
```

[3] https://raw.githubusercontent.com/JossRubio/Pytorch-UNet/patch-1/unet/unet$_m$odel.py

```
20
21  train_loss, val_loss = [], []
22
23  for epoch in range(epochs):
24      net.train()
25
26      epoch_loss = 0
27      with tqdm(total=n_train, desc=f'Epoch {epoch + 1}/{epochs}', unit='img') as pbar:
28          for batch in train_loader:
29              imgs = batch['image']
30              true_masks = batch['mask']
31              assert imgs.shape[1] == net.n_channels, \
32                  f'Network has been defined with {net.n_channels} input channels, ' \
33                  f'but loaded images have {imgs.shape[1]} channels. Please check that ' \
34                  'the images are loaded correctly.'
35
36              imgs = imgs.to(device=device, dtype=torch.float32)
37              mask_type = torch.float32 if net.n_classes == 1 else torch.long
38              true_masks = true_masks.to(device=device, dtype=mask_type)
39
40              masks_pred = net(imgs)
41              loss = criterion(masks_pred, true_masks)
42              epoch_loss += loss.item()
43              writer.add_scalar('Loss/train', loss.item(), global_step)
44
45              pbar.set_postfix(**{'loss (batch)': loss.item()})
46
47              optimizer.zero_grad()
48              loss.backward()
49              nn.utils.clip_grad_value_(net.parameters(), 0.1)
50              optimizer.step()
51
52              pbar.update(imgs.shape[0])
53              global_step += 1
54              if global_step % (n_train // (2 * batch_size)) == 0:
55                  for tag, value in net.named_parameters():
56                      tag = tag.replace('.', '/')
57                      writer.add_histogram('weights/' + tag, value.data.cpu().numpy(), global_step)
58                      writer.add_histogram('grads/' + tag, value.grad.data.cpu().numpy(), global_step)
59                  val_score = eval_net(net, val_loader, device)
60                  scheduler.step(val_score)
61                  writer.add_scalar('learning_rate', optimizer.param_groups[0]['lr'], global_step)
62                  if save_cp:
63                    torch.save(net.state_dict(),
64                    dir_checkpoint + f'CP_middle_epoch{epoch + 1}.pth')
65
66                  if net.n_classes > 1:
67                      logging.info('Validation cross entropy: {}'.format(val_score))
68                      writer.add_scalar('Loss/test', val_score, global_step)
69                  else:
70                      logging.info('Validation Dice Coeff: {}'.format(val_score))
```

```
71                    writer.add_scalar('Dice/test', val_score, global_step)
72
73                writer.add_images('images', imgs, global_step)
74                if net.n_classes == 1:
75                    writer.add_images('masks/true', true_masks, global_step)
76                    writer.add_images('masks/pred', torch.sigmoid(masks_pred) > 0.5, global_step)
77        train_loss.append(loss.item())
78        val_loss.append(val_score)
79        if save_cp:
80            torch.save(net.state_dict(),
81                       dir_checkpoint + f'CP_epoch{epoch + 1}.pth')
82            print(f'Checkpoint {epoch + 1} saved !')
83            logging.info(f'Checkpoint {epoch + 1} saved !')
84
85    writer.close()
```

Código 5: Función de entrenamiento adaptada.

Donde el entrenamiento se realizó mediante la siguiente celda.

```
1  net = UNet(n_channels=3, n_classes=12, bilinear=True)
2  net.cuda()
3  device = 'cuda'
4
5  try:
6    train_net(net=net,
7              device=device)
8  except KeyboardInterrupt:
9    torch.save(net.state_dict(), 'INTERRUPTED.pth')
10   logging.info('Saved interrupt')
11   try:
12     sys.exit(0)
13   except SystemExit:
14     os._exit(0)
```

Código 6: Ejecución del entrenamiento usando la función `train_net`

A grandes rasgos, esta función realiza el entrenamiento de la red utilizando las imagenes y mascaras especificadas en los parámetros `imgs_dir` y `masks_dir` respectivamente (línea 4), donde cargadas las imagenes mediante el objeto KittiDataset se separaron en conjunto de entrenamiento y validación mediante la función `DataLoader`. Teniendo esto, las imagenes fueron extraídas desde el Loader con un tamaño del **batch igual a 1**, donde para cada época se fueron guardando los valores del `loss` para el entrenamiento y la validación.

La parte más destacada de este desarrollo fueron los `checkpoints`, los cuales guardaron los valores de los parámetros para todas las épocas. Este registro se hizo mediante la función `torch.save`, donde el estado de la red iba siendo guardada con el comando `.state_dict` (línea 92 y 109).

Con el entrenamiento finalizado, y una vez obtenidos los valores `loss` para entrenamiento y validación entonces se graficaron las curvas de costes de ambos conjuntos (El código de obtención de las curvas se encuentra en el anexo).

2.5. Predicción.

La red `UNet` recibe tensores de dimensiones 1x3xHxW, donde las dimensiones de su salida son de 1x12xHxW. Dada la dimensionalidad de estos tensores entonces se les tuvo que hacer un proceso para poder visualizar el resultado de la red. Para ello se programó la siguiente función:

```
def predict_unet(net, train_eval):
  global batch_size, train_loader, test_loader
  if train_eval:
    data_loader = train_loader
    fig, axs = plt.subplots(5,3,figsize=(11, 11))
  else:
    data_loader = test_loader
    fig, axs = plt.subplots(5,2,figsize=(11, 11))

  for i, data in enumerate(data_loader):
    image = data['image']
    true_image = image.numpy().transpose(2,3,1,0)
    true_image = np.squeeze(true_image)
    if train_eval:
      mask = data['mask']
      true_mask =  mask.numpy().transpose(1,2,0)
      true_mask = np.squeeze(true_mask)
    predict_mask = net(image.cuda()).cpu().detach().numpy()
    predict_mask = np.squeeze(predict_mask.transpose(2,3,1,0))
    predict_mask = np.argmax(predict_mask, axis=2)
    axs[i, 0].imshow(true_image)
    axs[i, 0].set_title('True Image RGB')
    if train_eval:
      axs[i, 1].imshow(true_mask)
      axs[i, 1].set_title('True Mask')
      axs[i, 2].imshow(predict_mask)
      axs[i, 2].set_title('Predict_Mask')
    else:
      axs[i, 1].imshow(predict_mask)
      axs[i, 1].set_title('Predict_Mask')

    if i == 4:
      break
```

Código 7: Función de predicción visual, esta recibe variable externas (`batch_size`, `train_loader`, y `test_loader`)

Esta función recibe una red preentrenada, y un valor booleano que indica si se desea analizar el conjunto de entrenamiento o validación, donde transforma el tensor resultante de la red en un arreglo de numpy de dimensiones HxW1. Este algoritmo realiza las predicciones sobre 5 imágenes, donde muestra estas imágenes con sus predicciones y la máscaras que estas tuviesen. Dependiendo del valor del último parámetro se pueden obtener 15 o 10 imágenes, ya que el conjunto de prueba no tiene máscaras que mostrar.

3. Resultados y Análisis

Al ejecutar en orden los bloques anteriores se obtuvieron comportamientos y predicciones interesantes.

3.1. Evolución de la función de pérdida.

Al guardar los valores de la pérdida durante el entrenamiento y visualizarlo se obtuvo la siguiente figura:

Figura 2: Valor de la entropía cruzada durante el entrenamiento.

Se presenta una gran disparidad al principio del entrenamiento, mostrando un valor de pérdida más alto para el entrenamiento, sin embargo este este valor termina siendo menor a medida que avanza el entrenamiento. Otra característica destacable es que los valores de pérdida del conjunto de entrenamiento son mas dispersos que los del conjunto de validación, donde se aprecia más estabilidad en los valores de la pérdida del conjunto de validación.

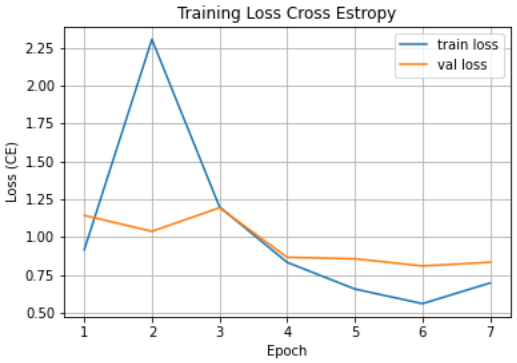

Figura 3: Comportamiento de las pérdidas al ejecutar nuevamente el entrenamiento, en un punto

El comportamiento mostrado en la figura 3 no se mantuvo al repetir el entrenamiento, sin embargo las características de las pérdidas de los conjuntos si lo hicieron (grandes cambios para la pérdida del conjunto de entrenamiento y posca desviación en los valores de pérdida de validación).

3.2. Predicciones del sistema, segunda época.

3.2.1. Conjunto de entrenamiento.

Al evaluar la red con los parámetros de la segunda época se obtuvieron las siguientes resultados:

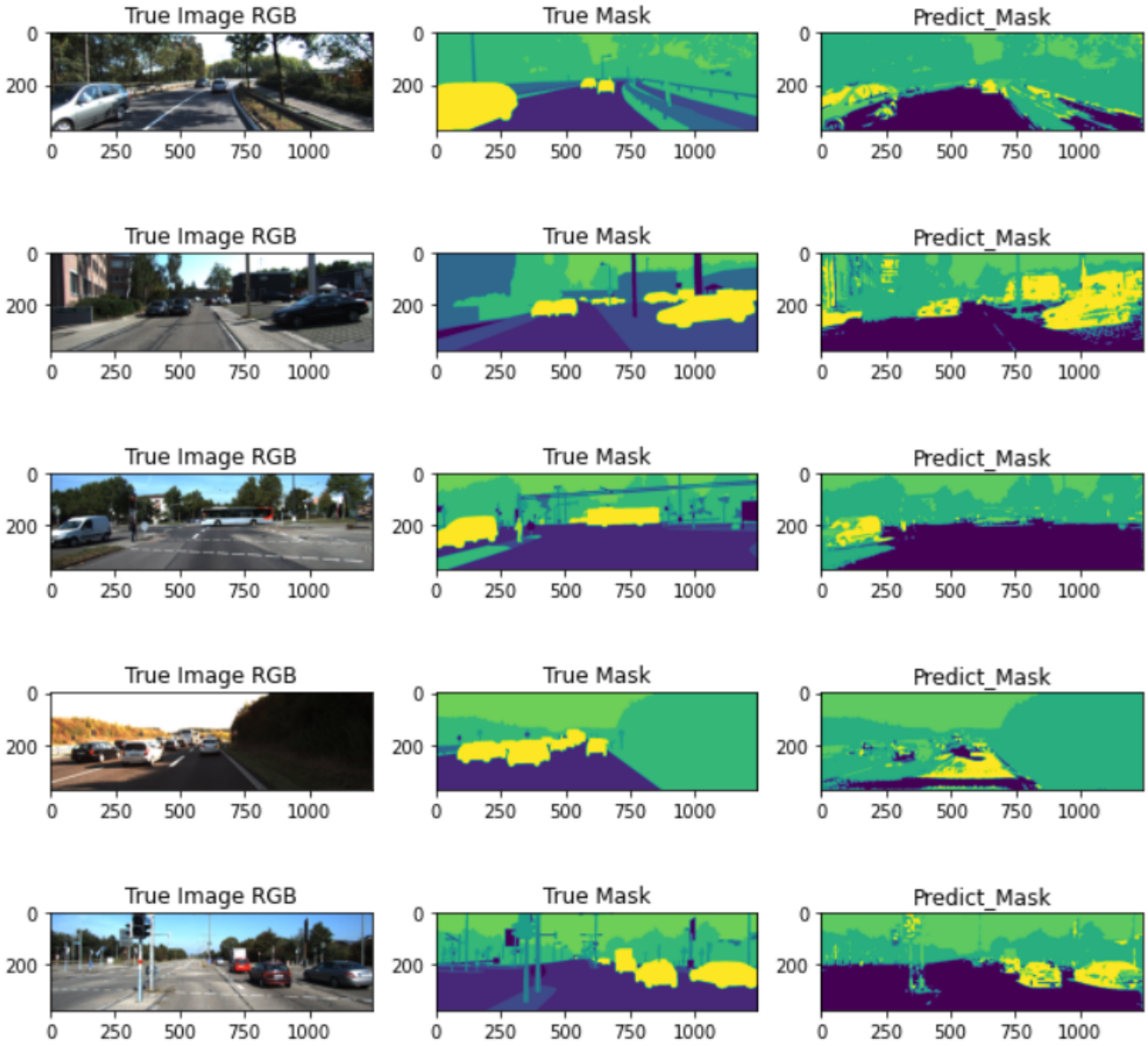

Figura 4: Imágenes, máscaras ground truth y predicciones realizadas con la los parámetros de la segunda época

En esta figura se aprecia que para todas las imagenes evaluadas no se tiene un gran acierto, mostrando grandes diferencias en la segmentación predicha al compararlas con la máscara ground truth.

3.2.2. Conjunto de prueba.

Utilizando los parámetros calculados a la segunda época, se obtuvieron las siguientes predicciones sobre el conjunto de test.

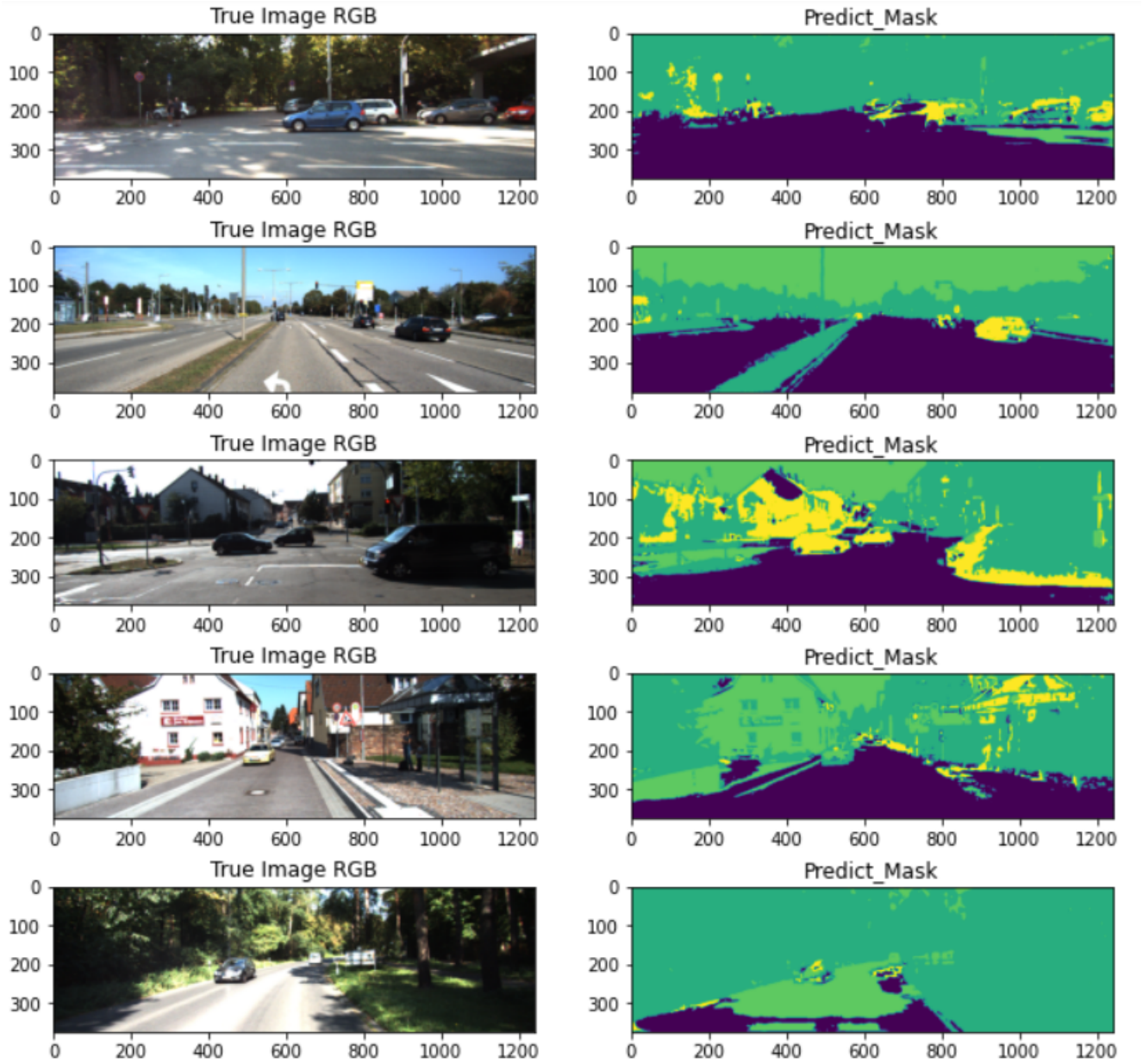

Figura 5: Segmentación semántica predicha para imágenes del conjunto de prueba.

Al igual que en el caso anterior, no se aprecia mucha precisión al generar las regiones de la segmentación, a pesar de que no se cuenten con máscaras ground truth para este conjunto, se aprecia que las regiones generadas no coinciden en su totalidad con los objetos presentes en la imagen original.

3.3. Predicciones del sistema, séptima época.

3.3.1. Conjunto de entrenamiento

Al finalizar el entrenamiento, se guardaron los valores de los parámetros como en el caso anterior, con eso se obtuvieron las siguientes predicciones:

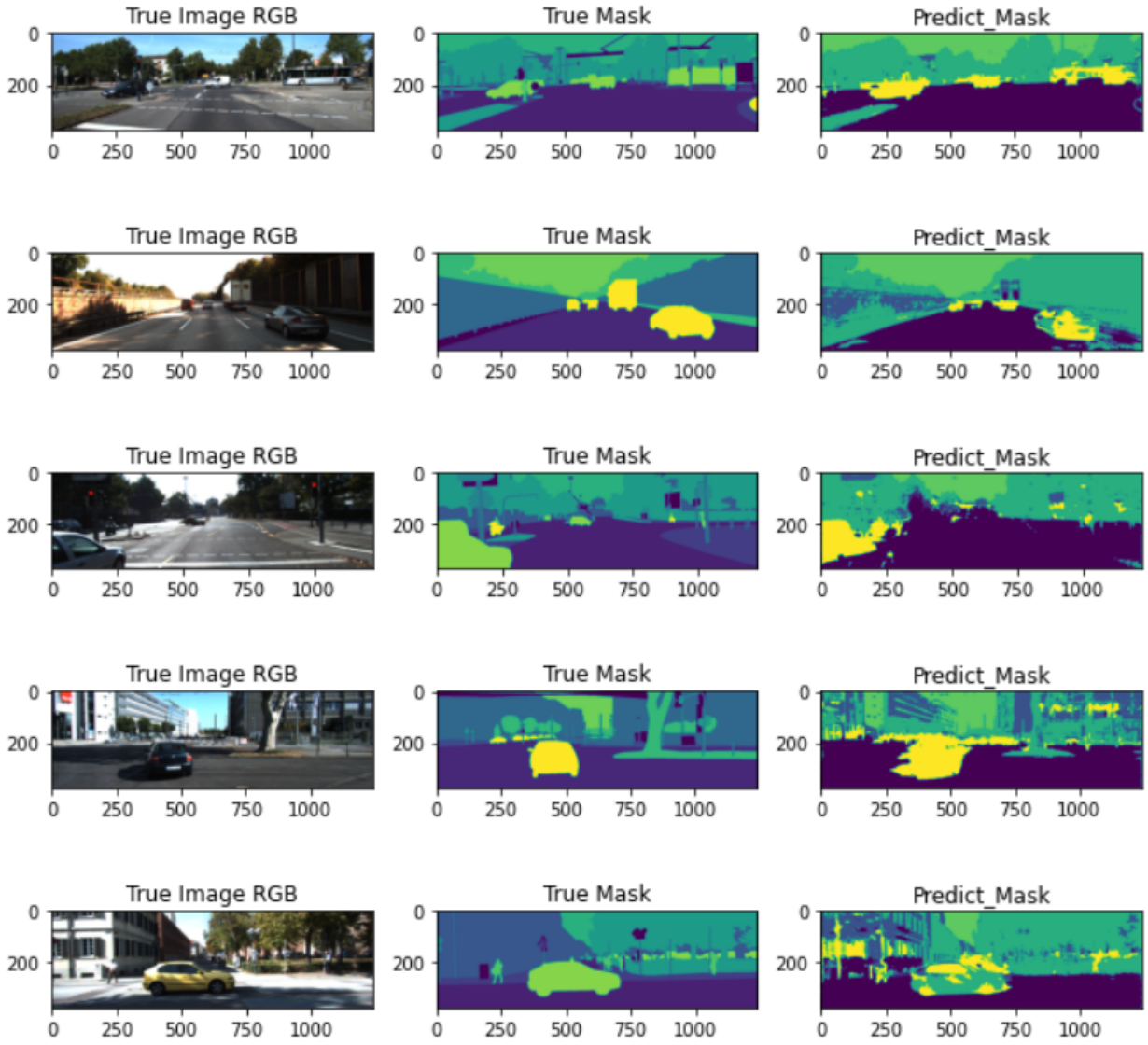

Figura 6: Imágenes, máscaras ground truth y predicciones realizadas con la los parámetros de la séptima época

Las segmentaciones obtenidas presentan regiones que no obedecen completamente a las máscaras con las que se les entrenó, sin embargo al comparar estas regiones obtenidas con las máscaras predichas de la figura 4, se aprecian que estas son un poco más precisas.

3.3.2. Conjunto de prueba.

Al evaluar con esta red sobre el conjunto de prueba se obtuvieron las siguientes predicciones.

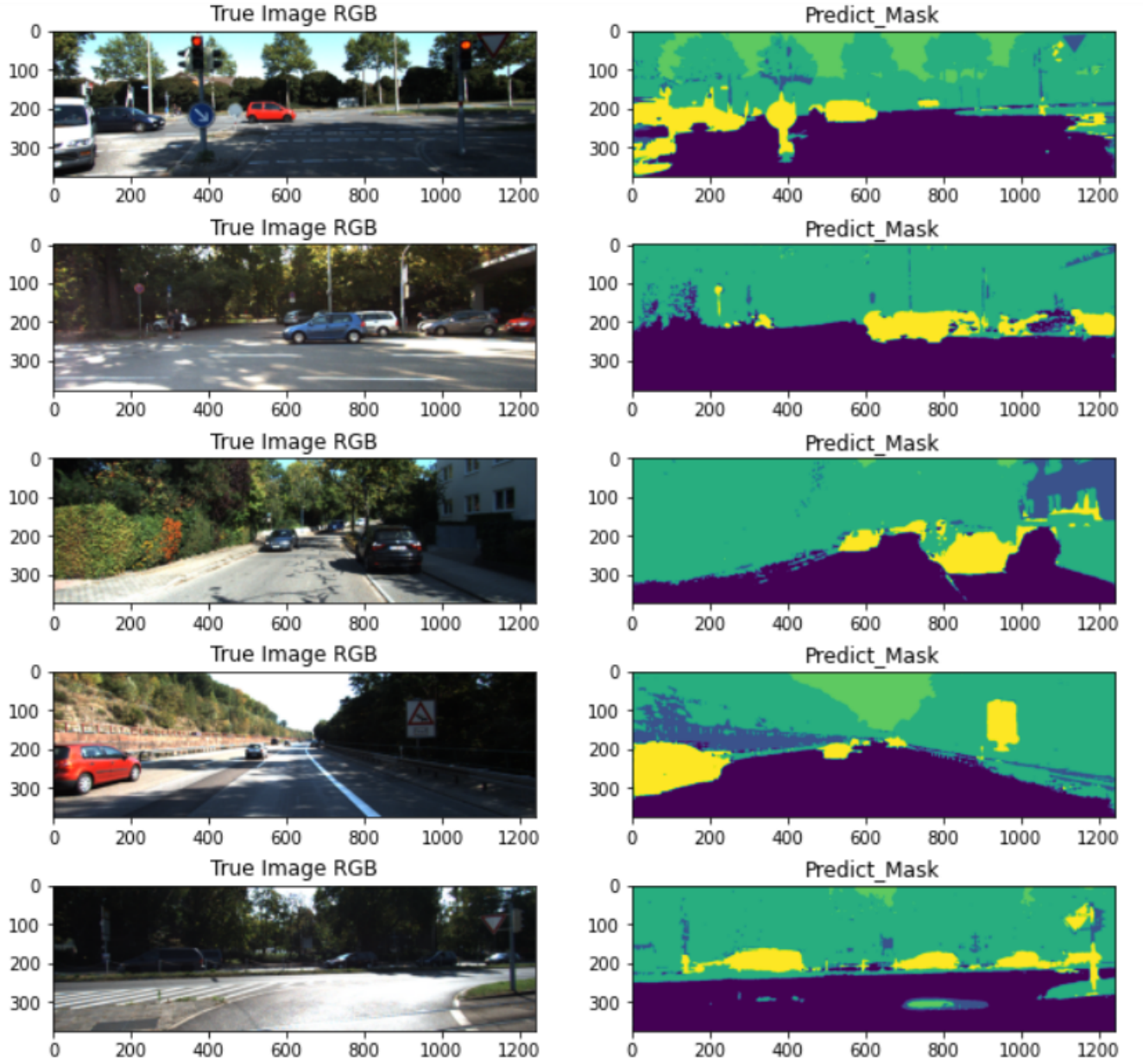

Figura 7: Segmentaciones semánticas predichas para el conjunto de prueba, con los parámetros de la red en la época 7.

Las predicciones realizadas por la red, al igual que en el conjunto de entrenamiento poseen una división que tiene mejores detecciones de objetos que la red entrenada hasta la segunda época, sin embargo esta lejos de realizar predicciones precisas, esto se aprecia en la tercera imagen de la figura anterior, donde las regiones que simbolizan la pista no esta claramente segmentadas, además que en la quinta figura también se tienen discontinuidades en estas mismas detecciones.

3.4. Análisis del desempeño del segmentador.

Las segmentación obtenida por el sistema esta lejos de ser precisa, sin embargo se lograron tener regiones notorias que simbolizaron los objetos presentes en las imágenes RGB.

Las pixeles que estuvieron más correctamente etiquetados fueron los que pertenecían a las regiones de los bordes, donde los colores de las regiones fueron los mismos para la mayoría de los casos evaluados, sin embargo al analizar las regiones que pertenecen a las pistas entonces se tienen problemas. La red utilizada tiene dificultades para asociar correctamente el color de la etiqueta para el asfalto, es capaz de separar relativamente correcto estas regiones, sin embargo el color de la etiqueta no es exactamente el mismo, teniendo una tonalidad más oscura para las predicciones al compararla con las máscaras originales. Este cambio puede ser debido al cambio de la escala de pixeles, además del poco entrenamiento que tuvo la red.

Otro objeto que pudo detectarse bien en la mayoría de los casos fueron los vehículos, aunque en algunas detecciones los pixeles de las regiones de los vehículos se mezclaron con los del fondo, aunque esta mezcla era algo esperable ya que en las máscaras ground truth se aprecia el cambio de tono del color para esos vehículos.

No se tuvo un rendimiento satisfactorio con esta red, aunque es posible debido a la pequeña cantidad de épocas con las que fue entrenado el sistema, se lograron aproximaciones satisfactorias de las segmentaciones pero no precisas del todo por lo que un modo de aumentar el rendimiento consistiría en aumentar la cantidad de épocas, o aplicar una reducción de clases.

4. Conclusión

La segmentación semántica es uno de los algoritmos de deep learning mas interesantes que hay debido a sus interesantes aplicaciones, por lo que el aprendizaje obtenido en esta experiencia fue muy valiosa. Si bien se reutilizó código de un repositorio de Github, también se programaron funciones importantes como la de predicción para obtener imágenes mostradas en este informe (figura 4, 5, 6, 7) se configuró correctamente para obtener los resultados obtenidos. Este proceso de adaptación permitió ahorrarse bastante trabajo y acelerar el proceso de comparación.

Debido a la reutilización de código, se logró descubrir nuevas funcionalidades de pytorch, un ejemplo de estos fue la función `torch.save()`, `torch.load()` para poder guardar el estado de la red a medida que se estuviese entrenando. Este fue un aprendizaje valiosos ya que debido a la gran cantidad de modelos de deep learning es necesario guardar una, red para que al compararla con otros sistemas se tengan los valores de los parámetros que le dieron el mejor rendimiento al sistema configurado.

Las principales dificultades encontradas durante el desarrollo de esta experiencias, se encontró en el mismo punto anterior. El analizar un código funcional no fue una tarea sencilla, aunque dentro de este código se utilizaron librerías con las que cuales ya se tenía algo de experiencias. Sin embargo el valor principal de de este análisis fue la lectura del programa mandado, ya que contó con varios comandos que permitieron estructurar de forma correcta el sistema.

Los rendimientos obtenidos de las redes estan lejos de ser perfectos, sin embargo este puede mejorar simplemente aumentado la cantidad de épocas del entrenamiento o cambiando levemente la arquitectura de la red colocando `Batch Normalization o dropout`, aunque es posible que esto no aumente mucho el rendimiento del sistema, podría reducir la diferencia entre las pérdidas del conjunto de entrenamiento con el conjunto de validación.

A pesar de que no se programó una gran parte del sistema utilizado, se adquirieron muchos conocimientos en programación al desarrollar este trabajo, debido al uso de nuevos comandos para cargar funciones y datos que estaban guardados ya sea en repositorios de Github, como en la web. Ademas como el enfoque de este trabajo fue el analizar visualmente la segmentación semántica hecha, entonces el desarrollo de la red no fue prioridad. Sin embargo esto no quita la posibilidad de probar otros sistemas para realizar este tipo de algoritmo, por ende uno de los trabajos futuros consistiría en la construcción del algún modelo que fuese distinto al UNet, pero que tuviese características similares debido a lo complejo que es el entrenamiento

Referencias

[1] *Segmentación*, Articulo de Wikipedia. Link: https://es.wikipedia.org/wiki/Segmentación_(procesamiento_de_imágenes)

[2] *Convolutional Mechanisms Semantic Segmentation*, Javier Ruiz del Solar, Presentación del curso EL7008-1 Procesamiento Avanzado de Imágenes. Semestre Primavera 2020.

[3] *Mask-RCNN* Javier Ruiz del Solar, Presentación del curso EL7008-1 Procesamiento Avanzado de Imágenes. Semestre Primavera 2020.

[4] *Saving and Load models* Matthew Inkawhich, Articulo de la página oficial de Pytorch. Link: https://pytorch.org/tutorials/beginner/saving_loading_models.html

5. Anexos.

5.1. Función kitty_inverse_map_1channel().

```
from numba import jit
@jit(nopython=True)
def kitty_inverse_map_1channel(img):
  cmap = [
    (0, 0), #void (ignorable)
    (4, 0),
    (5, 0),
    (6, 0),
    (7, 1), #road
    (8, 2), #sidewalk
    (9, 2),
    (10, 0), #rail truck (ignorable)
    (11, 3), #construction
    (12, 3),
    (13, 3),
    (14, 3),
    (15, 3),
    (16, 3),
    (17, 4), #pole(s)
    (18, 4),
    (19, 5), #traffic sign
    (20, 5),
    (21, 6), #vegetation
    (22, 6),
    (23, 7),
    (24, 8), #sky
    (25, 8),
    (26, 9), #human
    (27, 9),
    (28, 9),
    (29, 9),
    (30, 9),
    (31, 10), #train
    (32, 11), #cycle
    (33, 11)
  ]

  arrmap = np.zeros( (34), dtype=np.int32 )

  for el in cmap:
    arrmap[el[0]] = el[1]

  val = np.ones((img.shape[0],img.shape[1]), dtype=np.int32) * -1

  for i in range(img.shape[0]):
```

```
    for j in range(img.shape[1]):
      val[i,j] = arrmap[img[i,j]]
  return val
```

Código 8: Función para llevar las etiquetas de las máscaras de 31 valores a 12 labels posibles.

5.2. Red UNet.

```
""" Full assembly of the parts to form the complete network """

import torch.nn.functional as F

from unet_parts import *


class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=True):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inc = DoubleConv(n_channels, 64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        factor = 2 if bilinear else 1
        self.down4 = Down(512, 1024 // factor)
        self.up1 = Up(1024, 512 // factor, bilinear)
        self.up2 = Up(512, 256 // factor, bilinear)
        self.up3 = Up(256, 128 // factor, bilinear)
        self.up4 = Up(128, 64, bilinear)
        self.outc = OutConv(64, n_classes)

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        logits = self.outc(x)
        return logits
```

Código 9: Red UNet.

5.3. Curvas de pérdidas para entrenamiento y validación.

```
plt.plot(range(1, epochs+1), train_loss, label='train loss')
plt.plot(range(1, epochs+1), val_loss, label='val loss')
plt.title('Training Loss Cross Estropy')
plt.xlabel('Epoch')
plt.ylabel('Loss (CE)')
plt.grid(True)
plt.legend()
plt.savefig('CELoss_Train.png')
plt.show()
```

Código 10: Bloque para obtención de la evolución de las pérdidas

5.4. DataLoaders para predicciones.

```
val_percent = .1
img_scale = 1
batch_size = 1

dataset = KittiDataset(imgs_dir=dir_img, masks_dir=dir_mask, read_mask=True, scale=
    ↪ img_scale)
n_val = int(len(dataset) * val_percent)
n_train = len(dataset) - n_val
train, val = random_split(dataset, [n_train, n_val])
train_loader = DataLoader(train, batch_size=batch_size, shuffle=True, num_workers=8,
    ↪ pin_memory=True)
val_loader = DataLoader(val, batch_size=batch_size, shuffle=False, num_workers=8, pin_memory
    ↪ =True, drop_last=True)
```

Código 11: Variables preliminares para la función de predicción de la red