“Don't expect to be easy to work, because if you're easy to do, your heart will be flattered and arrogant.”
_ Buddha

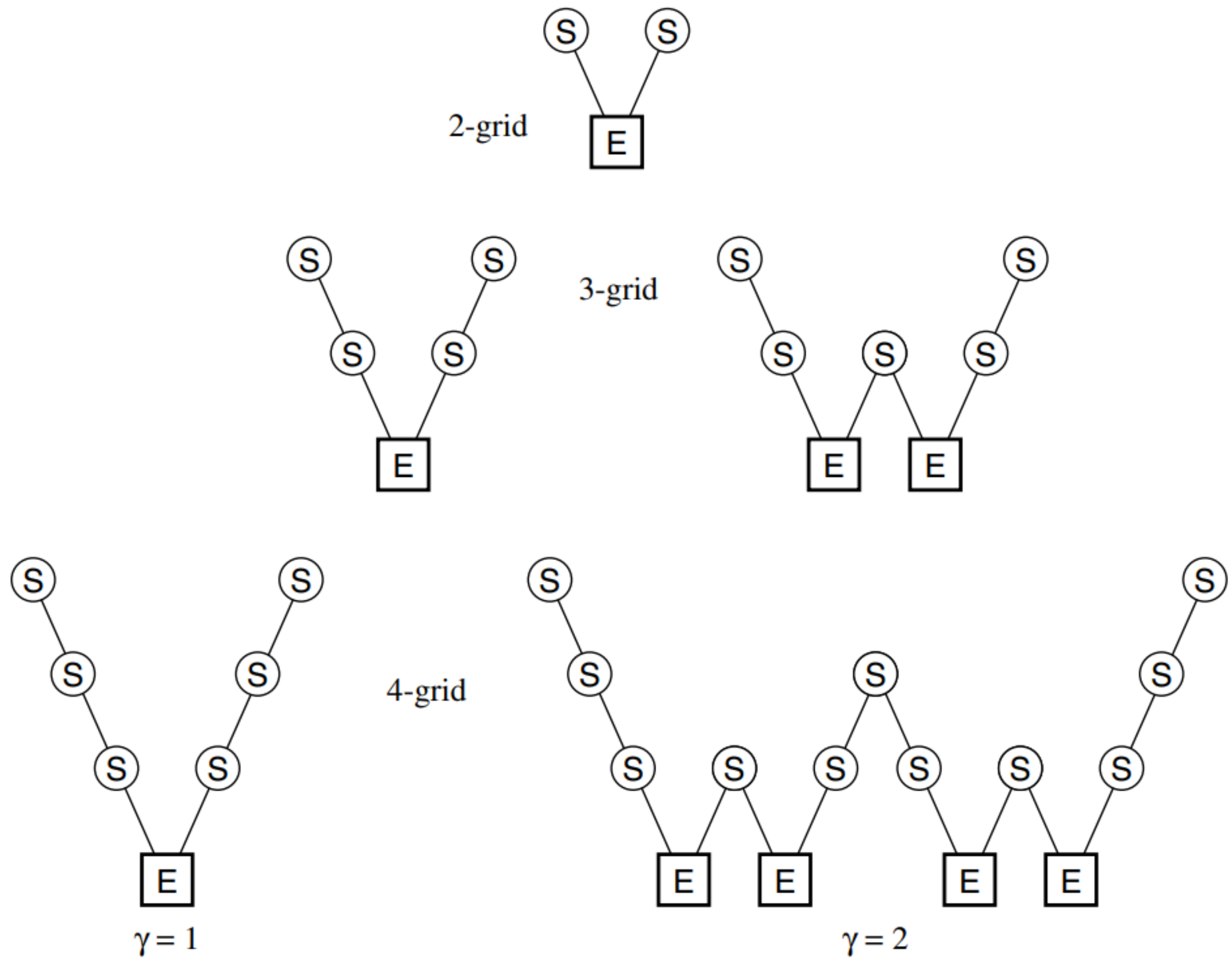
DIP: Multiscale Techniques

Presenter: Dr. Ha Viet Uyen Synh.

Part A

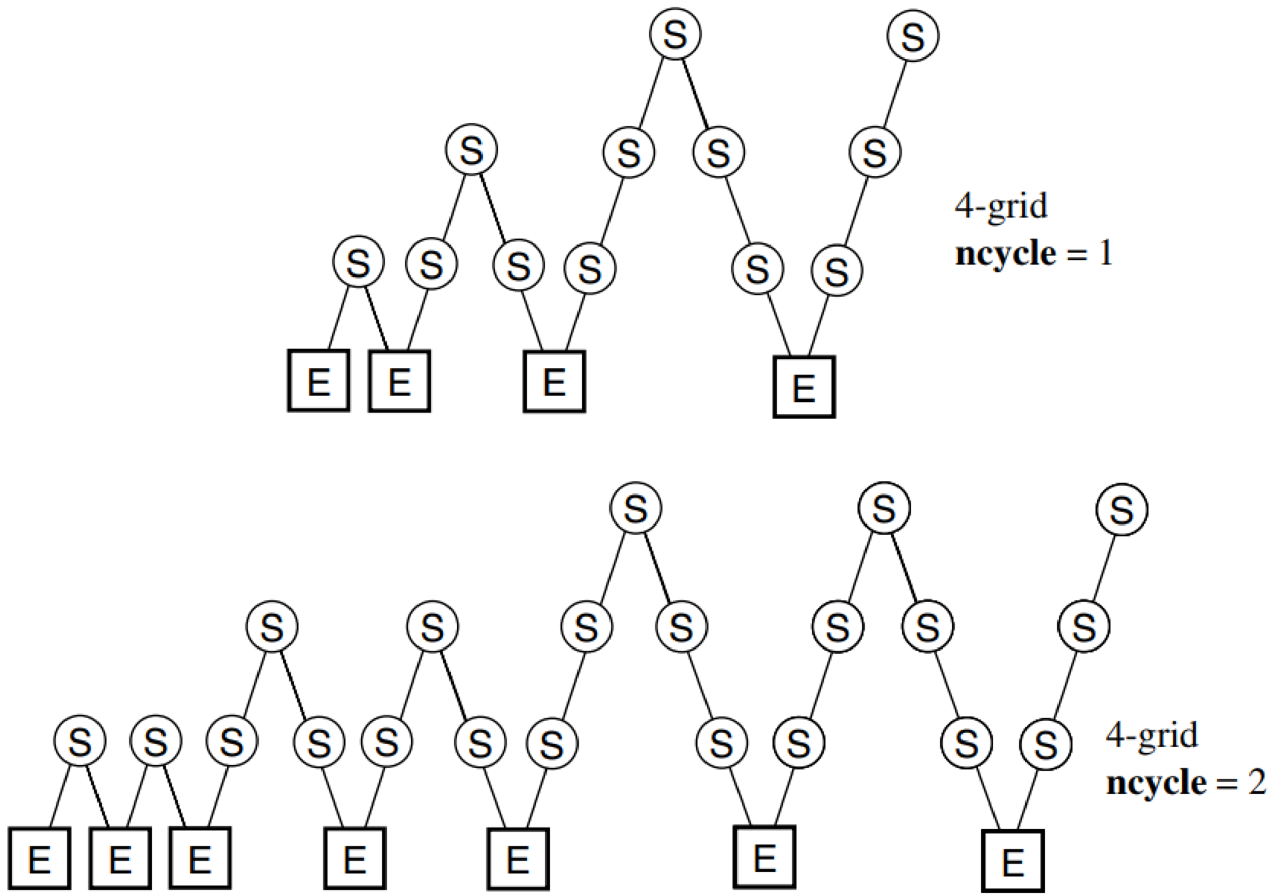
MULTIGRID TECHNIQUES

Structure of multigrid cycles



S denotes smoothing, while **E** denotes exact solution on the coarsest grid.

Structure of cycles for the full multigrid (FMG) method



Part C

POOLING LAYER

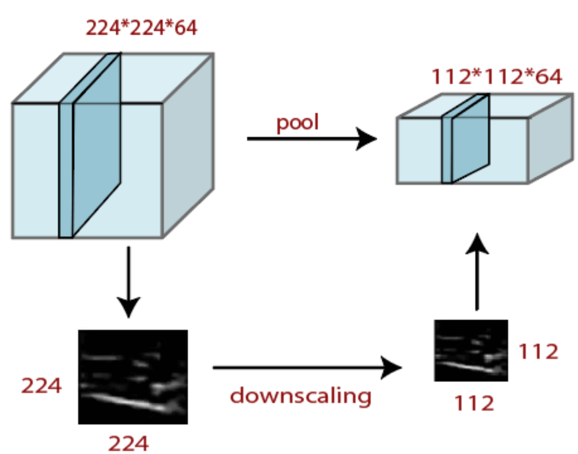
Pooling Layer

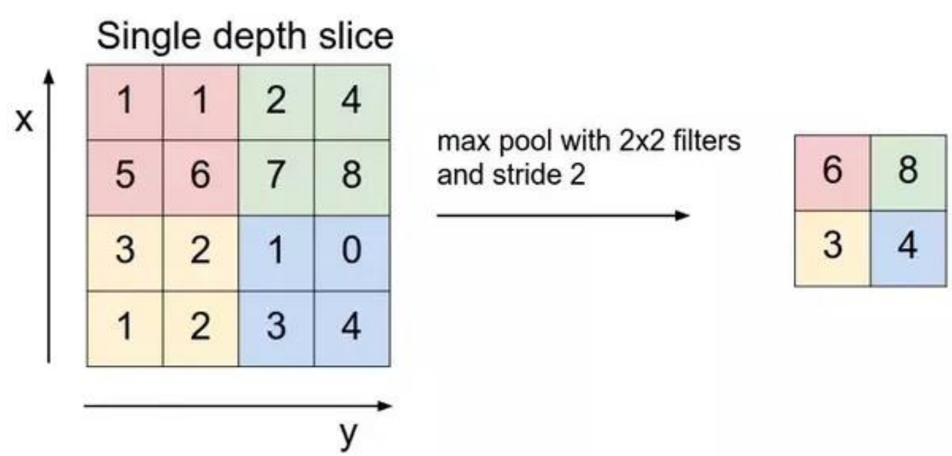
Pooling layers section would reduce the number of parameters when the images are too large. Spatial pooling also called subsampling or down sampling which reduces the dimensionality of each map but retains important information.

Spatial pooling can be of different types:

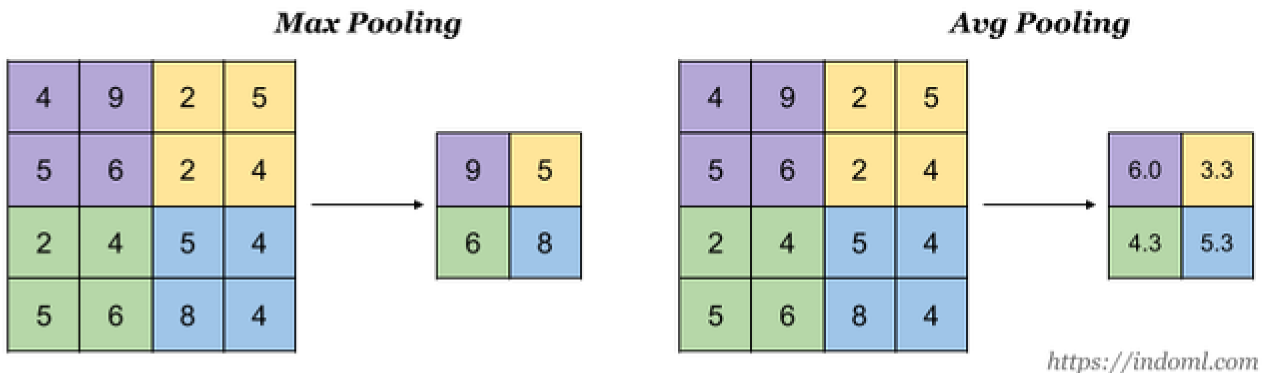
- Max Pooling
- Average Pooling
- Sum Pooling

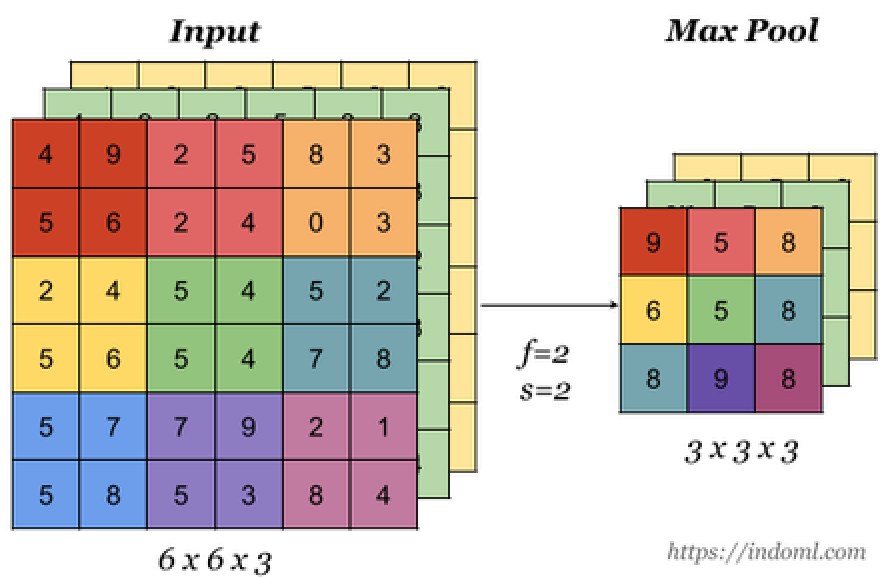
Max pooling takes the largest element from the rectified feature map. Taking the largest element could also take the average pooling. Sum of all elements in the feature map call as sum pooling.





Example


Max Pooling
4 9 2 5
5 6 2 4
2 4 5 4
5 6 8 4
9 5
6 8
Avg Pooling
4 9 2 5
5 6 2 4
2 4 5 4
5 6 8 4
6.0 3.3
4.3 5.3
https://indoml.com


Input
4 9 2 5 8 3
5 6 2 4 0 3
2 4 5 4 5 2
5 6 5 4 7 8
5 7 7 9 2 1
5 8 5 3 8 4
6 x 6 x 3
f=2
s=2
Max Pool
9 5 8
6 5 8
8 9 8
3 x 3 x 3
https://indoml.com

MaxPool2d

Syntax:

torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)

The parameters kernel_size, stride, padding, dilation can either be:

- a single int – in which case the same value is used for the height and width dimension
- a tuple of two ints – in which case, the first int is used for the height dimension, and the second int for the width dimension

- Input: (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}), where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 * \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 * \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

Example

```
# Import the required libraries
import torch
import torch.nn as nn
'''input of size = [N,C,H, W] or [C,H, W]
N==>batch size,
C==> number of channels,
H==> height of input planes in pixels,
W==> width in pixels.
'''
input = torch.empty(3, 4, 4).random_(256)
print("Input Tensor:", input)
print("Input Size:",input.size())
```

```
Input Tensor: tensor([
[[ 62., 215., 33., 70.],
[220., 228., 173., 81.],
[173., 117., 19., 90.],
[241., 63., 101., 218.]],

[[221., 85., 104., 69.],
[144., 63., 149., 187.],
[132., 252., 152., 211.],
[244., 5., 55., 191.]],

[[195., 127., 247., 175.],
[ 56., 61., 105., 72.],
[ 52., 233., 20., 147.],
[ 23., 184., 2., 114.]]])

Input Size: torch.Size([3, 4, 4])
```

```python
# pool of square window of size=3, stride=1
pooling1 = nn.MaxPool2d(3, stride=1)

# Perform Max Pool
output = pooling1(input)
print("Output Tensor:", output)
print("Output Size:",output.size())
```

```
Output Tensor: tensor([
[[228., 228.],
[241., 228.]],

[[252., 252.],
[252., 252.]],

[[247., 247.],
[233., 233.]]])

Output Size: torch.Size([3, 2, 2])
```

```python
# pool of non-square window
pooling2 = nn.MaxPool2d((2, 1), stride=(1, 2))
print("Output Kernel:",pooling2)
# Perform Max Pool
output = pooling2(input)
print("Output Tensor:", output)
print("Output Size:",output.size())
```

```
Output Kernel: MaxPool2d(kernel_size=(2, 1), stride=(1,
2), padding=0, dilation=1, ceil_mode=False)

Output Tensor: tensor([
[[220., 173.],
[220., 173.],
[241., 101.]],

[[221., 149.],
[144., 152.],
[244., 152.]],

[[195., 247.],
[ 56., 105.],
[ 52., 20.]]])

Output Size: torch.Size([3, 3, 2])
```

Example

```python
# Import the required libraries
import torch
import torchvision
from PIL import Image
import torchvision.transforms as T
import torch.nn.functional as F
# read the input image
img = Image.open('elephant.jpg')

# convert the image to torch tensor
img = T.ToTensor()(img)
print("Original size of Image:", img.size()) #Size([3, 466, 700])

# unsqueeze to make 4D
img = img.unsqueeze(0)

# define max pool with square window of size=4, stride=1
pool = torch.nn.MaxPool2d(4, 1)
img = pool(img)
img = img.squeeze(0)
print("Size after MaxPool:",img.size())
img = T.ToPILImage()(img)
img.show()
```

AvgPool2d

Syntax:

torch.nn.AvgPool2d(kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True, divisor_override=None)

The parameters kernel_size, stride, padding can either be:

- a single int – in which case the same value is used for the height and width dimension
- a tuple of two ints – in which case, the first int is used for the height dimension, and the second int for the width dimension

- Input: (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in}).
- Output: (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}), where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \right\rfloor$$

Example

```
# Import the required libraries
import torch
import torchvision
from PIL import Image
import torchvision.transforms as T
import torch.nn.functional as F

# read the input image
img = Image.open('panda.jpg')

# convert the image to torch tensor
img = T.ToTensor()(img)
print("Original size of Image:", img.size())#Size([3, 466, 700])

# unsqueeze to make 4D
img = img.unsqueeze(0)

# define avg pool with square window of size=4, stride=1
pool = torch.nn.AvgPool2d(4, 1)
img = pool(img)
img = img.squeeze(0)
print("Size after AvgPool:",img.size())
img = T.ToPILImage()(img)
img.show()
```



Part D

APPLICATIONS _ U-NET VS W-NET

What is U-Net?

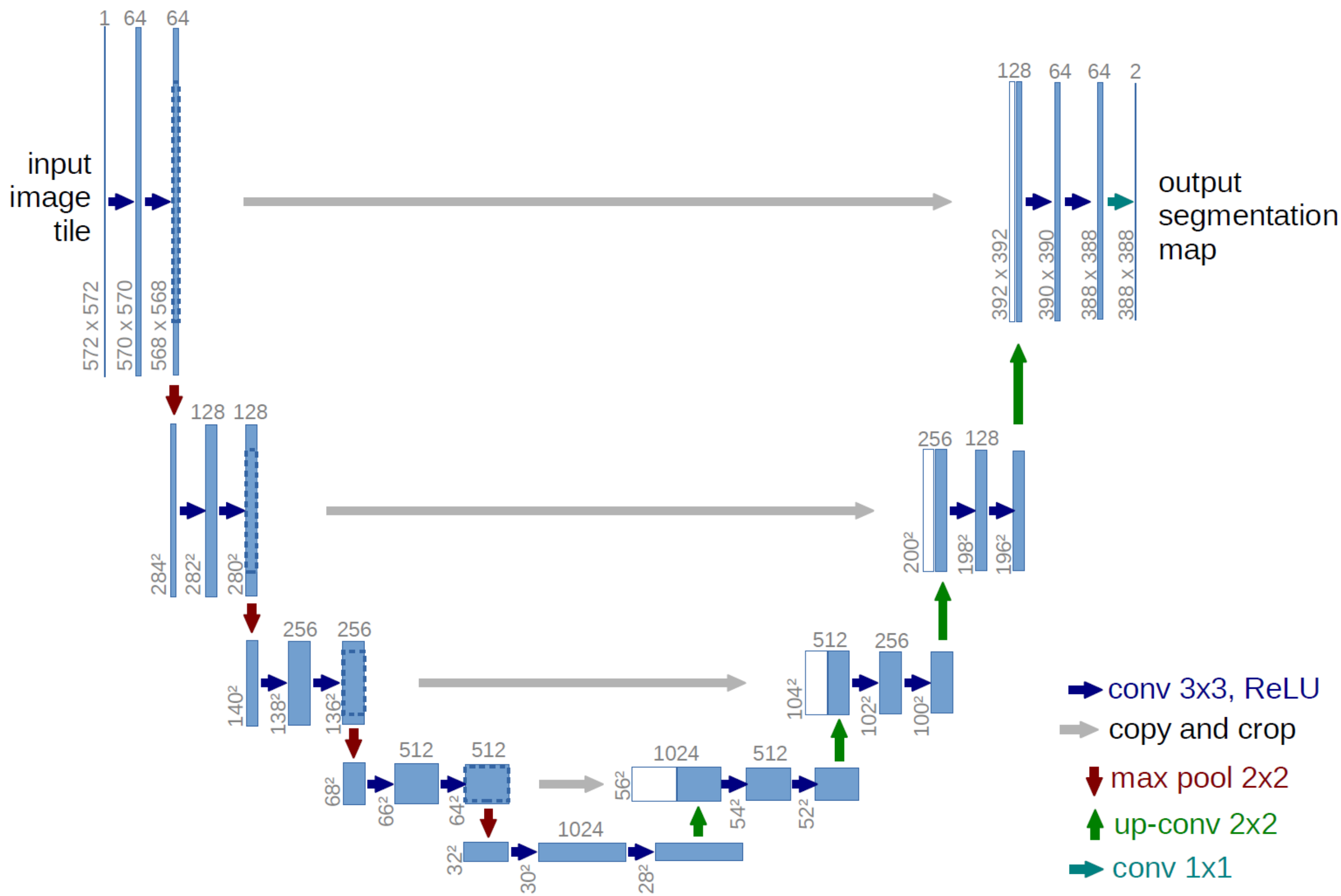
The U-NET was developed by Olaf Ronneberger et al. for Bio Medical Image Segmentation.

The architecture contains two paths.

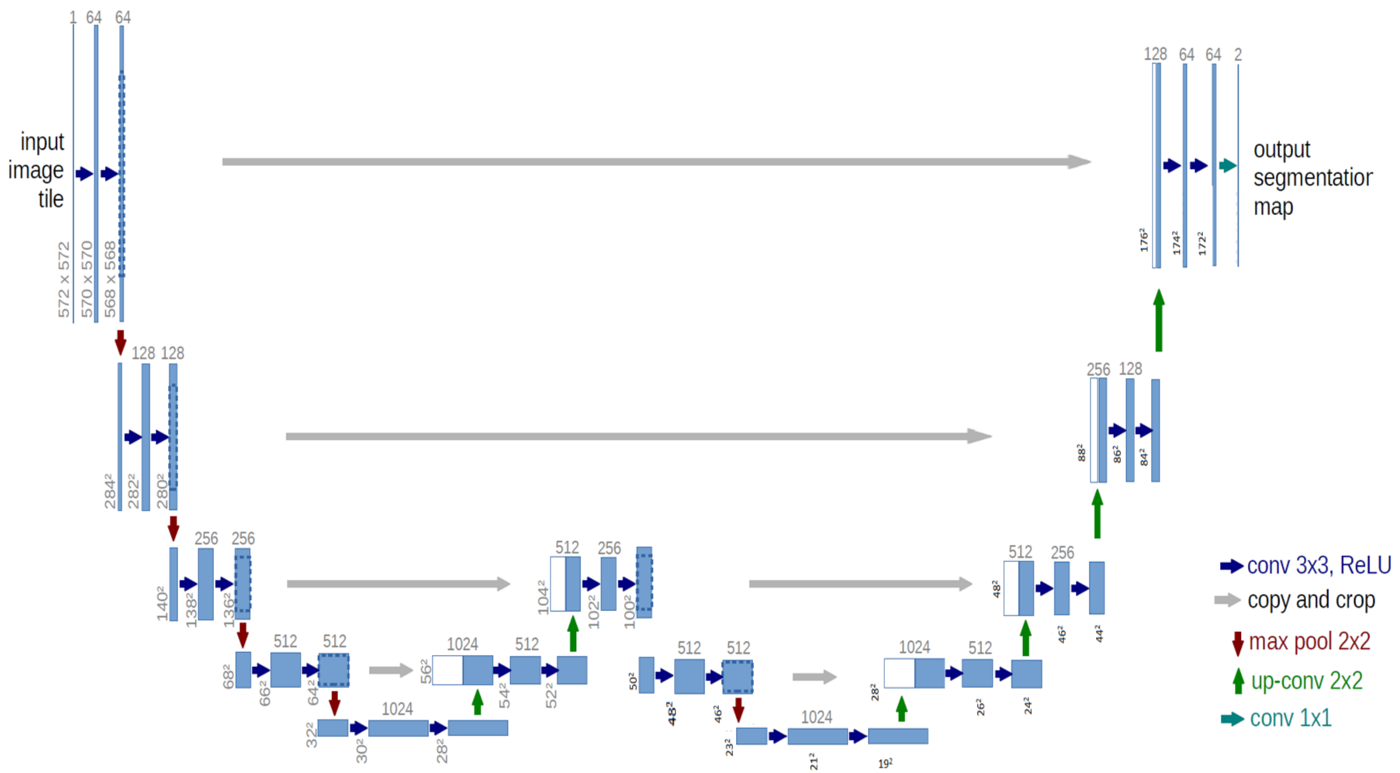
- The first path is the *contraction path* (also called as the encoder) which is used to capture the context in the image. The encoder is just a traditional stack of convolutional and max pooling layers.

- The second path is the *symmetric expanding path* (also called as the decoder) which is used to enable precise localization using transposed convolutions.

U-Net Segmentation

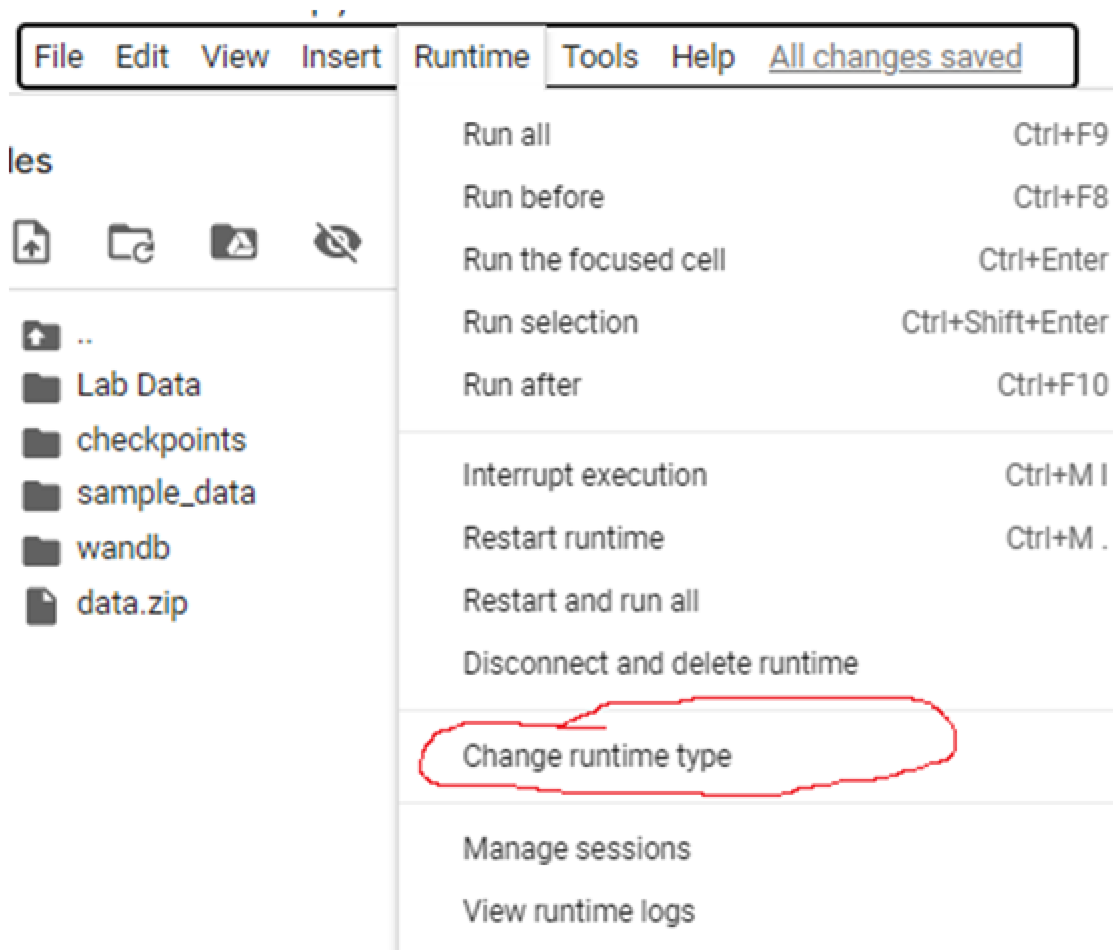


W-Net Segmentation

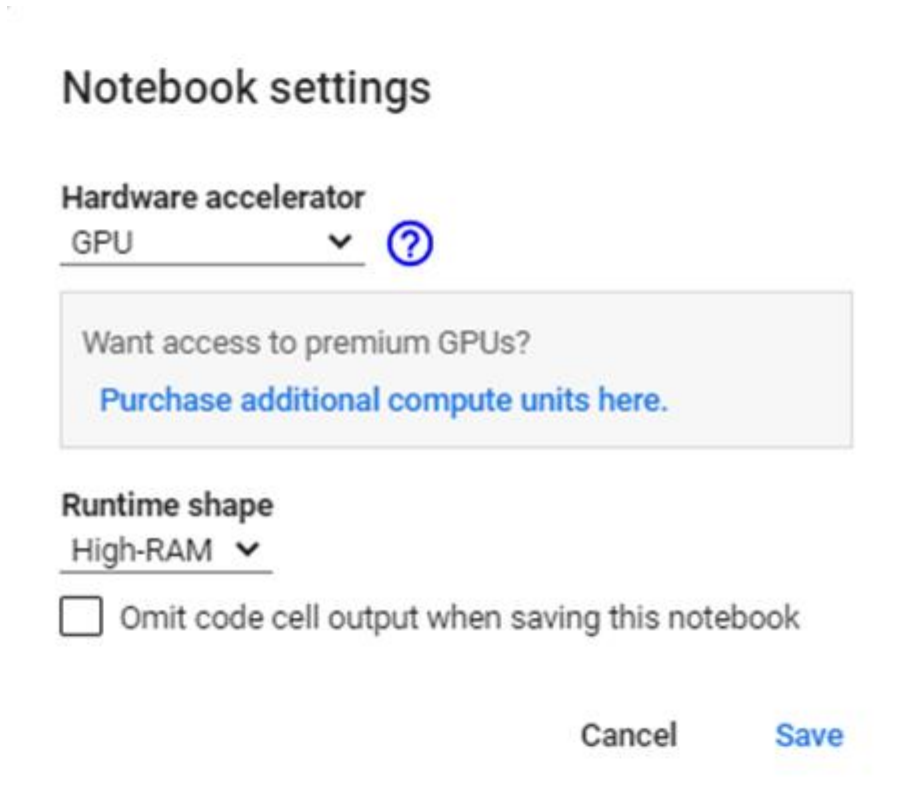


1. Set up environment for Google Colab

Runtime -> Change runtime type to enable GPU on GG colab notebook



Select GPU of the Hardware accelerator tab and Save notebook



2. Training Data - Carvana Image Masking Challenge

We select a subset of 544 images from the original dataset Carvana Image Masking Challenge . You can train the segmentation model with the entire data. However, the training process will take longer time.

This dataset contains a large number of car images (as .jpg files). Each car has exactly 16 images, each one taken at different angles. Each car has a unique id and images are named according to id_01.jpg, id_02.jpg … id_16.jpg. In addition to the images, you are also provided some basic metadata about the car make, model, year, and trim.

For the training set, you are provided a .gif file that contains the manually cutout mask for each image. The competition task is to automatically segment the cars in the images in the test set folder. To deter hand labeling, we have supplemented the test set with car images that are ignored in scoring.

Note 1: in Lab Data folder

imgs folder: this folder contains the training set images

masks folder: this folder contains the training set masks in .gif format

Note 2: For the other dataset, the format of mask images should be .gif format and binary value (0 and 255) if there are 2 classes

Load datasets

To compare the training process of 2 models: U-Net and W-Net, we use wandb library for visualization. You can create your own account if you want

```
!wget --load-
cookies /tmp/cookies.txt "https://docs.google.com/uc?expo
rt=download&confirm=$(wget --quiet --save-
cookies /tmp/cookies.txt --keep-session-cookies --no-
check-
certificate 'https://docs.google.com/uc?export=download&i
d=1Ju8sx3gpemfY3YILLs_ZNxrLF_05b-1b' -O- | sed -
rn 's/.*confirm=([0-9A-Za-
z_]+).*/\1\n/p')&id=1Ju8sx3gpemfY3YILLs_ZNxrLF_05b-1b" -
O data.zip && rm -rf /tmp/cookies.txt
!unzip /content/data.zip -d /content
```

```
!pip install wandb
```

Import the libraries needed

```
import logging
from os import listdir
from os.path import splitext
from pathlib import Path
import numpy as np
import torch
import wandb
from torch import Tensor
from PIL import Image
from torch.utils.data import Dataset
import torch.nn.functional as F
from tqdm import tqdm
import matplotlib.pyplot as plt
import cv2
import numpy as np
from torch import optim
from torch.utils.data import DataLoader, random_split
from tqdm import tqdm
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

Create the BasicDataset class

```
class BasicDataset(Dataset):
    def __init__(self, images_dir: str, masks_dir: str, scale: float = 1.0, mask_suffix: str = ''):
        self.images_dir = Path(images_dir)
        self.masks_dir = Path(masks_dir)
        assert 0 < scale <= 1, 'Scale must be between 0 and 1'
        self.scale = scale
        self.mask_suffix = mask_suffix
        self.ids = [splitext(file)[0] for file in listdir(images_dir) if not file.startswith('.')]
        if not self.ids:
            raise RuntimeError(f'No input file found in {images_dir}, make sure you put your images there')
        logging.info(f'Creating dataset with {len(self.ids)} examples')

    def __len__(self):
        return len(self.ids)
```

```python
    @staticmethod
    def preprocess(pil_img, scale, is_mask):
        w, h = pil_img.size
        newW, newH = int(scale * w), int(scale * h)
        assert newW > 0 and newH > 0, 'Scale is too small, resized images would have no pixel'
        pil_img = pil_img.resize((newW, newH), resample=Image.NEAREST if is_mask else Image.BICUBIC)
        img_ndarray = np.asarray(pil_img)

        if not is_mask:
            if img_ndarray.ndim == 2:
                img_ndarray = img_ndarray[np.newaxis, ...]
            else:
                img_ndarray = img_ndarray.transpose((2, 0, 1))

            img_ndarray = img_ndarray / 255

        return img_ndarray

    @staticmethod
    def load(filename):
        ext = splitext(filename)[1]
        if ext == '.npy':
            return Image.fromarray(np.load(filename))
        elif ext in ['.pt', '.pth']:
            return Image.fromarray(torch.load(filename).numpy())
        else:
            return Image.open(filename)
```

```
    def __getitem__(self, idx):
        name = self.ids[idx]
        mask_file = list(self.masks_dir.glob(name + self.mask_suffix +
 '.*'))
        img_file = list(self.images_dir.glob(name + '.*'))

        assert len(img_file) == 1, f'Either no image or multiple image
s found for the ID {name}: {img_file}'
        assert len(mask_file) == 1, f'Either no mask or multiple masks
 found for the ID {name}: {mask_file}'
        mask = self.load(mask_file[0])
        img = self.load(img_file[0])
        assert img.size == mask.size, \
            f'Image and mask {name} should be the same size, but are {
img.size} and {mask.size}'
        img = self.preprocess(img, self.scale, is_mask=False)
        mask = self.preprocess(mask, self.scale, is_mask=True)
        return {
            'image': torch.as_tensor(img.copy()).float().contiguous(),
            'mask': torch.as_tensor(mask.copy()).long().contiguous()
        }

class CarvanaDataset(BasicDataset):
    def __init__(self, images_dir, masks_dir, scale=1):
        super().__init__(images_dir, masks_dir, scale, mask_suffix='_m
ask')
```

Define Dice coefficient for training model

```
def dice_coeff(input: Tensor, target: Tensor, reduce_batch_first: bool = False, epsilon=1e-6):
    # Average of Dice coefficient for all batches, or for a single mask
    assert input.size() == target.size()
    if input.dim() == 2 and reduce_batch_first:
        raise ValueError(f'Dice: asked to reduce batch but got tensor without batch dimension (shape {input.shape})')

    if input.dim() == 2 or reduce_batch_first:
        inter = torch.dot(input.reshape(-1),target.reshape(-1))
        sets_sum = torch.sum(input) + torch.sum(target)
        if sets_sum.item() == 0:
            sets_sum = 2 * inter

        return (2 * inter + epsilon) / (sets_sum + epsilon)
    else:
        # compute and average metric for each batch element
        dice = 0
        for i in range(input.shape[0]):
            dice += dice_coeff(input[i, ...], target[i, ...])
        return dice / input.shape[0]
```

```python
def multiclass_dice_coeff(input: Tensor, target: Tensor,
reduce_batch_first: bool = False, epsilon=1e-6):
    # Average of Dice coefficient for all classes
    assert input.size() == target.size()
    dice = 0
    for channel in range(input.shape[1]):
        dice += dice_coeff(input[:, channel, ...], target
[:, channel, ...], reduce_batch_first, epsilon)

    return dice / input.shape[1]


def dice_loss(input: Tensor, target: Tensor, multiclass:
bool = False):
    # Dice loss (objective to minimize) between 0 and 1
    assert input.size() == target.size()
    fn = multiclass_dice_coeff if multiclass else dice_co
eff
    return 1 - fn(input, target, reduce_batch_first=True)
```

3. Build the main module

```python
class DoubleConv(nn.Module):
    """(convolution => [BN] => ReLU) * 2"""

    def __init__(self, in_channels, out_channels, mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.double_conv(x)
```

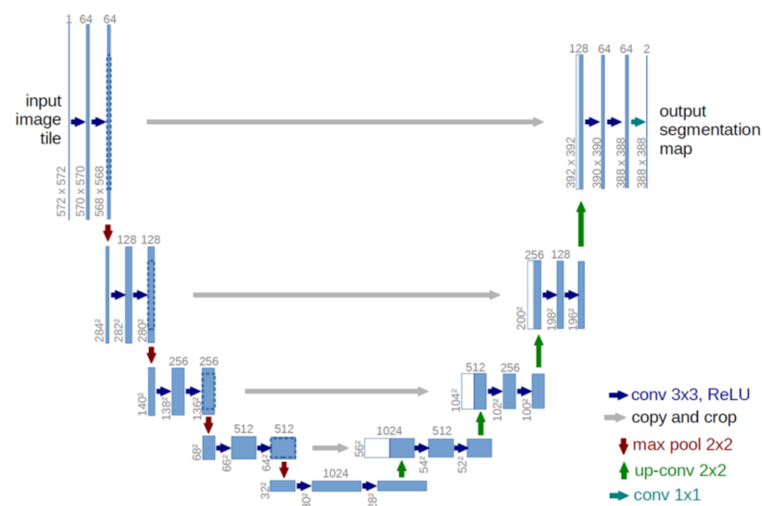
```python
class Down(nn.Module):
    """Downscaling with maxpool then double conv"""

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
        )

    def forward(self, x):
        return self.maxpool_conv(x)



class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels,
kernel_size=1)

    def forward(self, x):
        return self.conv(x)
```

```
class Up(nn.Module):
    """Upscaling then double conv"""
    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()

        # if bilinear, use the normal convolutions to reduce the number
 of channels
        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear', alig
n_corners=True)
            self.conv = DoubleConv(in_channels, out_channels, in_channe
ls // 2)
        else:
            self.up = nn.ConvTranspose2d(in_channels, in_channels // 2,
 kernel_size=2, stride=2)
            self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        # input is CHW
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
                        diffY // 2, diffY - diffY // 2])
        # if you have padding issues, see
        # https://github.com/HaiyongJiang/U-Net-Pytorch-Unstructured-
Buggy/commit/0e854509c2cea854e247a9c615f175f76fbb2e3a
        # https://github.com/xiaopeng-liao/Pytorch-
UNet/commit/8ebac70e633bac59fc22bb5195e513d5832fb3bd
        x = torch.cat([x2, x1], dim=1)
        return self.conv(x)
```

```python
def evaluate(net, dataloader, device):
    net.eval()
    num_val_batches = len(dataloader)
    dice_score = 0

    # iterate over the validation set
    for batch in tqdm(dataloader, total=num_val_batches, desc='Validation round', unit='batch', leave=False):
        image, mask_true = batch['image'], batch['mask']
        # move images and labels to correct device and type
        image = image.to(device=device, dtype=torch.float32)
        mask_true = mask_true.to(device=device, dtype=torch.long)
        mask_true = F.one_hot(mask_true, net.n_classes).permute(0, 3, 1, 2).float()

        with torch.no_grad():
            # predict the mask
            mask_pred = net(image)

            # convert to one-hot format
            if net.n_classes == 1:
                mask_pred = (F.sigmoid(mask_pred) > 0.5).float()
                # compute the Dice score
                dice_score += dice_coeff(mask_pred, mask_true, reduce_batch_first=False)
            else:
                mask_pred = F.one_hot(mask_pred.argmax(dim=1), net.n_classes).permute(0, 3, 1, 2).float()
                # compute the Dice score, ignoring background
                dice_score += multiclass_dice_coeff(mask_pred[:, 1:, ...], mask_true[:, 1:, ...], reduce_batch_first=False)


    net.train()

    # Fixes a potential division by zero error
    if num_val_batches == 0:
        return dice_score
    return dice_score / num_val_batches
```

4. Create the U-Net class

```python
class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=False):
        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
```



5. Build the Train_Net function

```
def train_net(net,
              device,
              epochs: int = 5,
              batch_size: int = 1,
              learning_rate: float = 1e-5,
              val_percent: float = 0.1,
              save_checkpoint: bool = True,
              img_scale: float = 0.5,
              amp: bool = False):
    # 1. Create dataset
    try:
        dataset = CarvanaDataset(dir_img, dir_mask, img_scale)
    except (AssertionError, RuntimeError):
        dataset = BasicDataset(dir_img, dir_mask, img_scale)

    # 2. Split into train / validation partitions
    n_val = int(len(dataset) * val_percent)
    n_train = len(dataset) - n_val
    train_set, val_set = random_split(dataset, [n_train, n_val],
generator=torch.Generator().manual_seed(0))
```

```python
    # 3. Create data loaders
    loader_args = dict(batch_size=batch_size, num_workers=4, pin_memory=True)
    train_loader = DataLoader(train_set, shuffle=True, **loader_args)
    val_loader = DataLoader(val_set, shuffle=False, drop_last=True, **loader_args)

    # (Initialize logging)
    experiment = wandb.init(project='Unet-Lab', anonymous='must')
    experiment.config.update(dict(epochs=epochs, batch_size=batch_size, learning_rate=learning_rate,
val_percent=val_percent, save_checkpoint=save_checkpoint, img_scale=img_scale, amp=amp))

    logging.info(f'''Starting training:
        Epochs:          {epochs}
        Batch size:      {batch_size}
        Learning rate:   {learning_rate}
        Training size:   {n_train}
        Validation size: {n_val}
        Checkpoints:     {save_checkpoint}
        Device:          {device.type}
        Images scaling:  {img_scale}
        Mixed Precision: {amp}
    ''')
```

```python
    # 4. Set up the optimizer, the loss, the learning rate scheduler and the loss scaling for AMP
    optimizer = optim.RMSprop(net.parameters(), lr=learning_rate, weight_decay=1e-8, momentum=0.9)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'max', patience=2)  # goal: maximize Dice score
    grad_scaler = torch.cuda.amp.GradScaler(enabled=amp)
    criterion = nn.CrossEntropyLoss()
    global_step = 0

    # 5. Begin training
    for epoch in range(1, epochs+1):
        net.train()
        epoch_loss = 0
        with tqdm(total=n_train, desc=f'Epoch {epoch}/{epochs}', unit='img') as pbar:
            for batch in train_loader:
                images = batch['image']
                true_masks = batch['mask']

                assert images.shape[1] == net.n_channels, \
                    f'Network has been defined with {net.n_channels} input channels, but loaded images have {images.shape[1]} channels. Please check that the images are loaded correctly.'

                images = images.to(device=device, dtype=torch.float32)
                true_masks = true_masks.to(device=device, dtype=torch.long)
```

```
                with torch.cuda.amp.autocast(enabled=amp):
                    masks_pred = net(images)
                    loss = criterion(masks_pred, true_masks) \
                           + dice_loss(F.softmax(masks_pred, dim=
1).float(), F.one_hot(true_masks, net.n_classes).permute(0, 3, 1,
 2).float(),  multiclass=True)

                optimizer.zero_grad(set_to_none=True)
                grad_scaler.scale(loss).backward()
                grad_scaler.step(optimizer)
                grad_scaler.update()

                pbar.update(images.shape[0])
                global_step += 1
                epoch_loss += loss.item()
                experiment.log({
                    'train loss': loss.item(),
                    'step': global_step,
                    'epoch': epoch
                })
                pbar.set_postfix(**{'loss (batch)': loss.item()})
```

```
                # Evaluation round
                division_step = (n_train // (10 * batch_size))
                if division_step > 0:
                    if global_step % division_step == 0:
                        histograms = {}
                        for tag, value in net.named_parameters():
                            tag = tag.replace('/', '.')
                            if not torch.isinf(value).any():
                                histograms['Weights/' + tag] = wandb.Hi
stogram(value.data.cpu())
                            if not torch.isinf(value.grad).any():
                                histograms['Gradients/' + tag] = wandb.
Histogram(value.grad.data.cpu())

                        val_score = evaluate(net, val_loader, device)
                        scheduler.step(val_score)

                        logging.info('Validation Dice score: {}'.format
(val_score))
                        experiment.log({
                            'learning rate': optimizer.param_groups[0][
'lr’], validation Dice': val_score, 'images':
wandb.Image(images[0].cpu()), 'masks': {
'true': wandb.Image(true_masks[0].float().cpu()),
'pred': wandb.Image(masks_pred.argmax(dim=1)[0].float().cpu()),},
'step': global_step, 'epoch': epoch, **histograms})
```

```python
if save_checkpoint:
    Path(dir_checkpoint).mkdir(parents=True, exist_ok=True)
    torch.save(net.state_dict(), str(dir_checkpoint / 'checkpoint_epoch{}.pth'.format(epoch)))
    logging.info(f'Checkpoint {epoch} saved!')
```

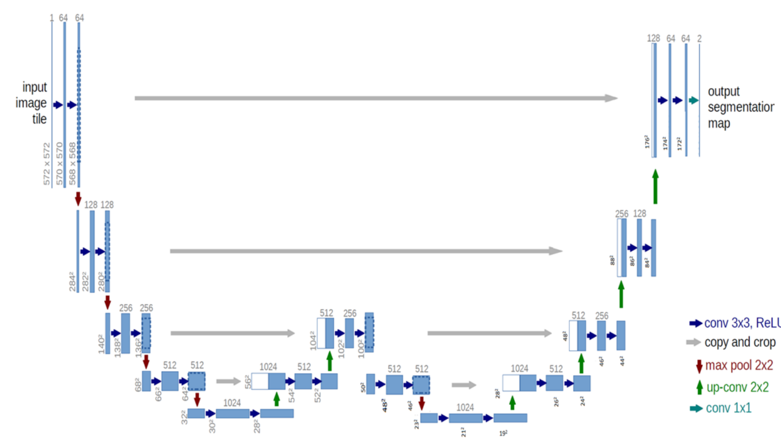
6. Parameter for Training U-Net Model

```
dir_checkpoint = Path('./checkpoints_Unet/')
dir_img = Path('./Lab Data/imgs/')
dir_mask = Path('./Lab Data/masks/’)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
epochs = 5
batch_size = 1
lr = 1e-5
scale = 0.5
val = 10
amp = False
classes = 2
bilinear = False

logging.basicConfig(level=logging.INFO, format='%(levelname)s: %(message)s')
logging.info(f'Using device {device}’)

net = UNet(n_channels=3, n_classes= classes, bilinear= bilinear)

logging.info(f'Network:\n'
             f'\t{net.n_channels} input channels\n'
             f'\t{net.n_classes} output channels (classes)\n'
             f'\t{"Bilinear" if net.bilinear else "Transposed conv"} upscaling')

net.to(device=device)
```

```python
try:
    train_net(net=net,
              epochs= epochs,
              batch_size= batch_size,
              learning_rate= lr,
              device=device,
              img_scale= scale,
              val_percent= val / 100,
              amp= amp)
except KeyboardInterrupt:
    torch.save(net.state_dict(), 'INTERRUPTED.pth')
    logging.info('Saved interrupt')
    raise
```

7. Create the W_Net class

```
class WNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=False):
        super(WNet, 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)

    # Wnet #####
    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_up1 = self.up1(x5, x4)
        x_up2 = self.up2(x_up1, x3)

        x4_4 = self.down3(x_up2)
        x5_5 = self.down4(x4_4)

        x_up11 = self.up1(x5_5, x4_4)
        x_up22 = self.up2(x_up11, x_up2)
        x_up33 = self.up3(x_up22, x2)
        x_up44 = self.up4(x_up33, x1)
        logits = self.outc(x_up44)
        return logits
```



8. Parameter for training W_Net Model

```
dir_checkpoint = Path('./checkpoints_Wnet/')

dir_img = Path('./Lab Data/imgs/')
dir_mask = Path('./Lab Data/masks/’)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
epochs = 5
batch_size = 1
lr = 1e-5
scale = 0.5
val = 10
amp = False
classes = 2
bilinear = False
```

9. Build the Train_WNet function

```
def train_Wnet(net,
               device,
               epochs: int = 5,
               batch_size: int = 1,
               learning_rate: float = 1e-5,
               val_percent: float = 0.1,
               save_checkpoint: bool = True,
               img_scale: float = 0.5,
               amp: bool = False):
    # 1. Create dataset
    try:
        dataset = CarvanaDataset(dir_img, dir_mask, img_scale)
    except (AssertionError, RuntimeError):
        dataset = BasicDataset(dir_img, dir_mask, img_scale)

    # 2. Split into train / validation partitions
    n_val = int(len(dataset) * val_percent)
    n_train = len(dataset) - n_val
    train_set, val_set = random_split(dataset, [n_train, n_val],
generator=torch.Generator().manual_seed(0))
```

```python
    # 3. Create data loaders
    loader_args = dict(batch_size=batch_size, num_workers=4, pin_memory=True)
    train_loader = DataLoader(train_set, shuffle=True, **loader_args)
    val_loader = DataLoader(val_set, shuffle=False, drop_last=True, **loader_args)

    # (Initialize logging)
    experiment = wandb.init(project='Wnet-Lab', resume='allow', anonymous='must')
    experiment.config.update(dict(epochs=epochs, batch_size=batch_size, learning_rate=learning_rate,
                                  val_percent=val_percent, save_checkpoint=save_checkpoint, img_scale=img_scale,
                                  amp=amp))

    logging.info(f'''Starting training:
        Epochs:          {epochs}
        Batch size:      {batch_size}
        Learning rate:   {learning_rate}
        Training size:   {n_train}
        Validation size: {n_val}
        Checkpoints:     {save_checkpoint}
        Device:          {device.type}
        Images scaling:  {img_scale}
        Mixed Precision: {amp}
    ''')
```

```
    # 4. Set up the optimizer, the loss, the learning rate scheduler and the loss scaling for AMP
    optimizer = optim.RMSprop(net.parameters(), lr=learning_rate, weight_decay=1e-8, momentum=0.9)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'max', patience=2)  # goal: maximize Dice score
    grad_scaler = torch.cuda.amp.GradScaler(enabled=amp)
    criterion = nn.CrossEntropyLoss()
    global_step = 0
```

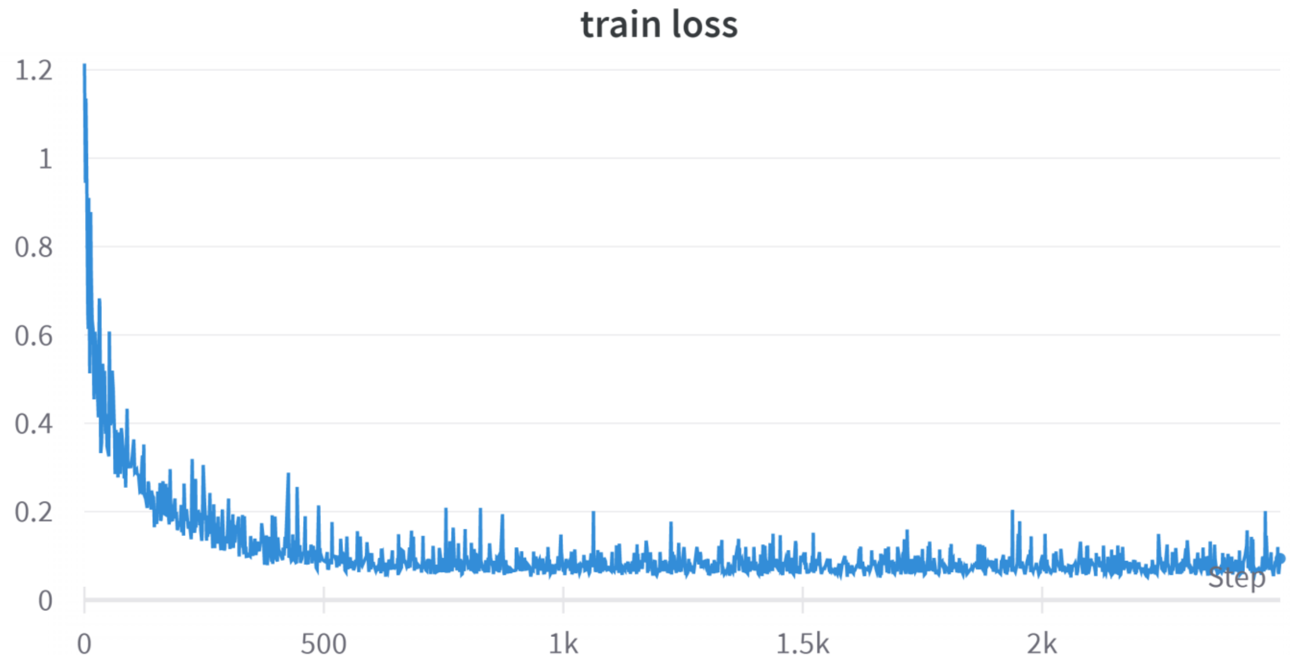
```python
    # 5. Begin training
    for epoch in range(1, epochs+1):
        net.train()
        epoch_loss = 0
        with tqdm(total=n_train, desc=f'Epoch {epoch}/{epochs}', unit='img') as pbar:
            for batch in train_loader:
                images = batch['image']
                true_masks = batch['mask']

                assert images.shape[1] == net.n_channels, \
                    f'Network has been defined with {net.n_channels} input channels, but loaded images have {images.shape[1]} channels. Please check that the images are loaded correctly.'
                images = images.to(device=device, dtype=torch.float32)
                true_masks = true_masks.to(device=device, dtype=torch.long)

                with torch.cuda.amp.autocast(enabled=amp):
                    masks_pred = net(images)
                    loss = criterion(masks_pred, true_masks) \
                           + dice_loss(F.softmax(masks_pred, dim=1).float(), F.one_hot(true_masks, net.n_classes).permute(0, 3, 1, 2).float(), multiclass=True)
```

```python
optimizer.zero_grad(set_to_none=True)
grad_scaler.scale(loss).backward()
grad_scaler.step(optimizer)
grad_scaler.update()

pbar.update(images.shape[0])
global_step += 1
epoch_loss += loss.item()
experiment.log({
    'train loss': loss.item(),
    'step': global_step,
    'epoch': epoch
})
pbar.set_postfix(**{'loss (batch)': loss.item()})
```

```
                # Evaluation round
                division_step = (n_train // (10 * batch_size))
                if division_step > 0:
                    if global_step % division_step == 0:
                        histograms = {}
                        for tag, value in net.named_parameters():
                            tag = tag.replace('/', '.')
                            if not torch.isinf(value).any():
                                histograms['Weights/' + tag] = wandb.Histogram
(value.data.cpu())
                            if not torch.isinf(value.grad).any():
                                histograms['Gradients/' + tag] = wandb.Histogr
am(value.grad.data.cpu())

                        val_score = evaluate(net, val_loader, device)
                        scheduler.step(val_score)

                        logging.info('Validation Dice score: {}'.format(val_sc
ore))
                        experiment.log({
                            'learning rate': optimizer.param_groups[0]['lr'],
                            'validation Dice': val_score,
                            'images': wandb.Image(images[0].cpu()),
                            'masks': {
                                'true': wandb.Image(true_masks[0].float().cpu(
)),
                                'pred': wandb.Image(masks_pred.argmax(dim=1)[0
].float().cpu()),
                            },
                            'step': global_step,
                            'epoch': epoch,
                            **histograms
                        })
        if save_checkpoint:
            Path(dir_checkpoint).mkdir(parents=True, exist_ok=True)
            torch.save(net.state_dict(), str(dir_checkpoint / 'checkpoint_epoc
h{}.pth'.format(epoch)))
            logging.info(f'Checkpoint {epoch} saved!')
```

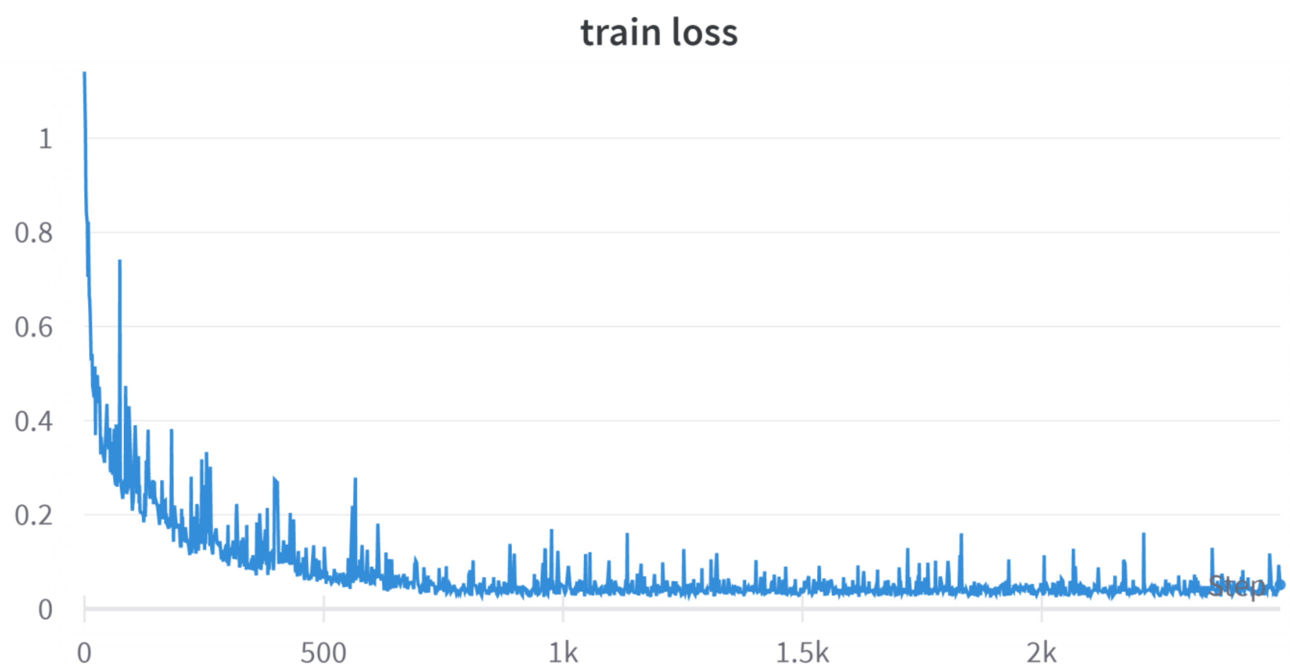
```
logging.basicConfig(level=logging.INFO, format='%(levelname)s: %(
message)s')
logging.info(f'Using device {device}')

net = WNet(n_channels=3, n_classes= classes, bilinear= bilinear)


logging.info(f'Network:\n'
                 f'\t{net.n_channels} input channels\n'
                 f'\t{net.n_classes} output channels (classes)\n'
                 f'\t{"Bilinear" if net.bilinear else "Transposed
 conv"} upscaling’)

net.to(device=device)

try:
    train_Wnet(net=net,
               epochs= epochs,
               batch_size= batch_size,
               learning_rate= lr,
               device=device,
               img_scale= scale,
               val_percent= val / 100,
               amp= amp)
except KeyboardInterrupt:
    torch.save(net.state_dict(), 'INTERRUPTED.pth')
    logging.info('Saved interrupt')
    raise
```

U-Net



train loss
1.2
1
0.8
0.6
0.4
0.2
0
Step
0
500
1k
1.5k
2k

W-Net



train loss
1
0.8
0.6
0.4
0.2
0
Step
0
500
1k
1.5k
2k

10. Predict image

```
!wget --load-cookies /tmp/cookies.txt "https://docs.google.com/uc?export=download&confirm=$(wget --quiet --save-cookies /tmp/cookies.txt --keep-session-cookies --no-check-certificate 'https://docs.google.com/uc?export=download&id=1KC90UwsY31OXMPq9mtFVRHesVX1zjpAG' -O- | sed -rn 's/.*confirm=([0-9A-Za-z_]+).*/\1\n/p')&id=1KC90UwsY31OXMPq9mtFVRHesVX1zjpAG" -O data.zip && rm -rf /tmp/cookies.txt
!unzip /content/data.zip -d /content
```

```python
import argparse
import logging
import os

import numpy as np
import torch
import torch.nn.functional as F
from PIL import Image
from torchvision import transforms
```

```python
def predict_img(net,
                full_img,
                device,
                scale_factor=1,
                out_threshold=0.5):
    net.eval()
    img = torch.from_numpy(BasicDataset.preprocess(full_img, scale_factor, is_mask=False))
    img = img.unsqueeze(0)
    img = img.to(device=device, dtype=torch.float32)

    with torch.no_grad():
        output = net(img)
        if net.n_classes > 1:
            probs = F.softmax(output, dim=1)[0]
        else:
            probs = torch.sigmoid(output)[0]

        tf = transforms.Compose([
            transforms.ToPILImage(),
            transforms.Resize((full_img.size[1], full_img.size[0])),
            transforms.ToTensor()
        ])

        full_mask = tf(probs.cpu()).squeeze()

    if net.n_classes == 1:
        return (full_mask > out_threshold).numpy()
    else:
        return F.one_hot(full_mask.argmax(dim=0), net.n_classes).permute(2, 0, 1).numpy()
```

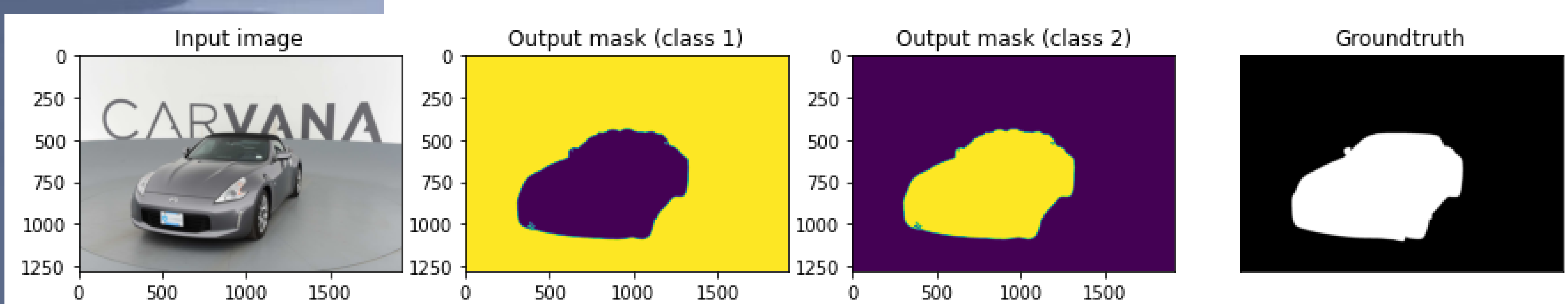
```python
# checkpoint_dir = "/content/checkpoints_Unet/checkpoint_epoch5.pth"
checkpoint_dir = "/content/checkpoints/checkpoint_epoch5.pth"

# image = "/content/predict/6ae670e86620_06.jpg"
image = "/content/predict/6ba36af67cb0_02.jpg"
# image = "/content/predict/6c0cd487abcd_10.jpg"
# image = "/content/predict/6c3470c34408_06.jpg"

# mask = "/content/predict/6ae670e86620_06_mask.gif"
mask = "/content/predict/6ba36af67cb0_02_mask.gif"
# mask = "/content/predict/6c0cd487abcd_10_mask.gif"
# mask = "/content/predict/6c3470c34408_06_mask.gif"


scale = 0.5
```

Visualize the prediction

```
def plot_img_and_mask(img, mask, groundTruth):
    classes = mask.shape[0] if len(mask.shape) > 2 else 1
    fig, ax = plt.subplots(1, classes + 2, figsize=(15, 15))

    ax[0].set_title('Input image')
    ax[0].imshow(img)
    if classes > 1:
        for i in range(classes):
            ax[i + 1].set_title(f'Output mask (class {i + 1})')
            ax[i + 1].imshow(mask[i, :, :])
    else:
        ax[1].set_title(f'Output mask')
        ax[1].imshow(mask)
    ax[3].set_title(f'Groundtruth')
    ax[3].imshow(groundTruth)

    plt.xticks([]), plt.yticks([])
    plt.show()
```

Create and load the trained model

```python
## Wnet ###
net = WNet(n_channels=3, n_classes=2, bilinear=False)

## Unet - Uncomment here ###
# net = UNet(n_channels=3, n_classes=2, bilinear=False)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
net.to(device=device)
net.load_state_dict(torch.load(checkpoint_dir, map_location=device))

img = Image.open(image)
groundtruth_mask = Image.open(mask)

mask = predict_img(net=net, full_img=img, scale_factor= scale, out_threshold
= 0.5, device=device)

plot_img_and_mask(img, mask, groundtruth_mask)
```



Any Questions?