Code adapted from https://github.com/maidacundo/real-time-fire-segmentation-deep-learning/tree/main

updated superclass() fix to correct node ordering and added a custom CUDA tensornode to further accelerate FPS.

In [1]:
```
# Settings for autoreloading.

%load_ext autoreload
%autoreload 2
```

In [2]:
```
import numpy as np
```

In [3]:
```
# Settings for reproducibility.

from src.utils.seed import set_random_seed

SEED = 42
set_random_seed(SEED)
```

In [4]:
```
from torch import cuda

# Set the device on which the model operations are performed.
DEVICE = 'cuda' if cuda.is_available() else 'cpu'
DEVICE
```

Out[4]:
```
'cuda'
```

In [5]:
```
BASE_DIR = '.'
```

Data Preparation

In [6]:
```
ORIGINAL_SHAPE = (3840, 2160)
RESIZE_SHAPE = (512, 512)
```

Data Extraction

In this section the images and masks data is extracted from the respective zip files and resized from size 3840×2160 to the input shape of the network 512×512. Each image is a matrix of the given shape containing *3 channels* (B, G, R) presenting values within the interval $[0, 255]$. On the other hand, a mask is a matrix of the given size, containing values binary values (0 or 1). The matrices representing the images and the masks are defined as:

- images: $\mathbb{N}_{[0, 255]}^{512 \times 512 \times 3}$.
- masks: $\mathbb{N}_{[0, 1]}^{512 \times 512}$.

In particular, each image represent a frame of a aerial video taken by a drone, representing a fire in an outdoor area. The respective masks segment semantically the fire from the background.

Images and masks are represented as numpy arrays.

In [7]:
```
import os
from src.data.dataset_handler import load_images_from_zip

images = load_images_from_zip(os.path.join(r"C:\Users\mvppr\Desktop\comms\SpecialTopic
                              are_masks=False,
                              resize_shape=RESIZE_SHAPE)
```

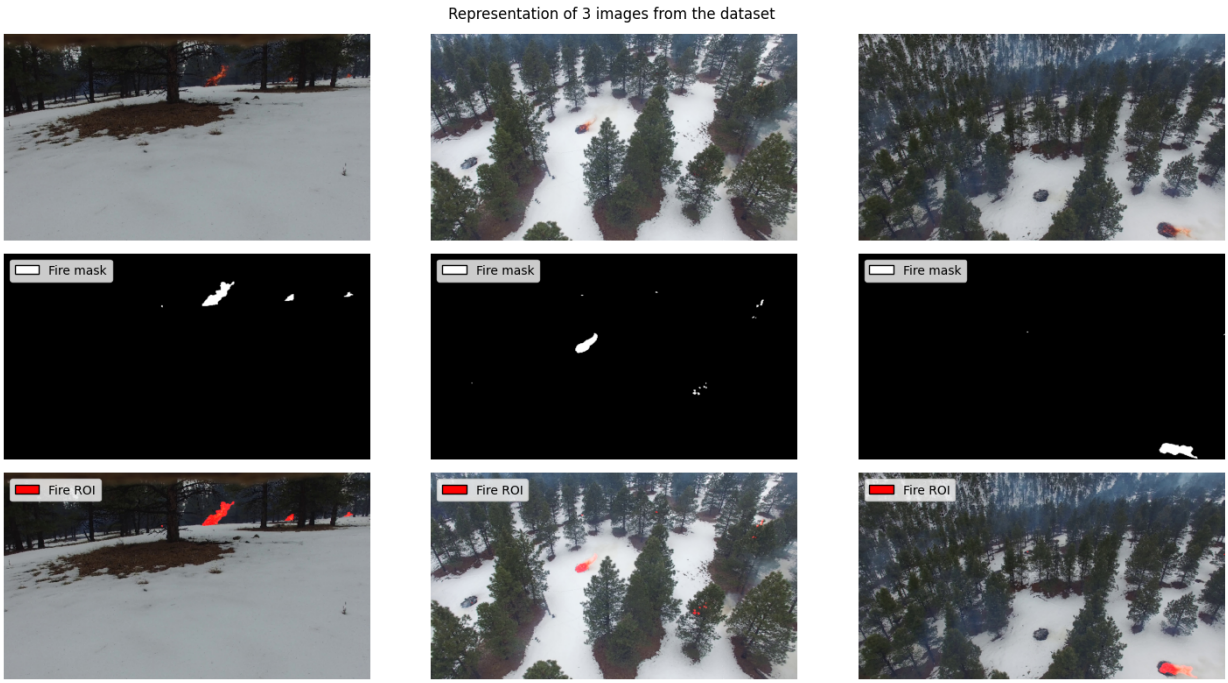
```
100%|██████████| 2003/2003 [03:57<00:00,  8.43it/s]
```

In [8]:
```
import os
from src.data.dataset_handler import load_images_from_zip

masks = load_images_from_zip(os.path.join(r"C:\Users\mvppr\Desktop\comms\SpecialTopicR
                             are_masks=True,
                             resize_shape=RESIZE_SHAPE)
```

```
100%|██████████| 2003/2003 [00:48<00:00, 40.92it/s]
```

In [9]:
```
# Assert that the number of images corresponds to the number of masks.
assert len(images) == len(masks), \
    'The number of images does not correspond to the number of masks.'
```

In [10]:
```
from src.data.analysis import plot_dataset_samples

plot_dataset_samples(images, masks, resize_shape=ORIGINAL_SHAPE)
```



Data split

According to the original paper, the dataset is split considering as test data the 15% of the whole dataset.

The rest of the data is used for the training procedure, specifically 85% of it for training the network and 15% for validating the results.

The data is shuffled while splitting according to the original seed, considering each instance indipendent to the other.

In [11]:
```
from src.data.dataset_handler import get_train_val_test_dataset_split

(X_train, y_train), (X_val, y_val), (X_test, y_test) = \
    get_train_val_test_dataset_split(images, masks, seed=SEED)
```

In [12]:
```
print(f'Train shapes: {X_train.shape}, {y_train.shape}')
print(f'Validation shapes: {X_val.shape}, {y_val.shape}')
print(f'Test shapes: {X_test.shape}, {y_test.shape}')
```

```
Train shapes: (1446, 512, 512, 3), (1446, 512, 512)
Validation shapes: (256, 512, 512, 3), (256, 512, 512)
Test shapes: (301, 512, 512, 3), (301, 512, 512)
```

Model Definition

The network is a modification of the *DeepLabV3+* model which is a model for segmantic segmentation composed of an encoder and a decoder. The encoder extracts features from the input image, and the decoder upsamples the features to the original image size. The applied modification serve to improve the speed of the segmentation process, in order to guarantee its usage for real-time applications. In particular this implementation refears to the semantic segmentation of aerial images captured by drone of woodland fire.

Encoder

The encoder is composed of the sequence of a *DCNN* backbone and a *ASPP* module.

DCNN

The *DCNN* backbone is composed of a series of convolutional layers. The first layer is a standard convolutional layer with 16 convolutional filters. This layer is followed by 15 *MobileNetV3* bottlenecks. The *MobileNetV3* bottlenecks are divided into different stages according to the size of the input features. In some *MobileNetV3* bottlenecks, the authors also use the *Squeeze-and-*

Excitation (SE) module. The *SE* module helps to improve the performance of the model by recalibrating the importance of each feature map.

Two different nonlinear activation functions are used in the *MobileNetV3* bottleneck: Rectified Linear Unit (*ReLU*) and Hard versions of Swish (*H-swish*).

The DCNN backbone outputs three intermediate feature maps f_1, f_2, and f_3. They are the results of 1^{st}, 3^{rd} and 6^{th} *MobileNetV3* bottleneck layers respectively.

Moreover, the result of the whole *DCNN* module is fed to the following *ASPP* module.

ASPP

Atrous Spatial Pyramid Pooling (ASPP) is a multi-scale feature aggregation module that is inspired by spatial pyramid pooling (SPP). *ASPP* effectively captures multi-scale contextual information by applying four atrous convolutions with different dilation rates in parallel on the deep feature output generated by the *DCNN*.

ASPP is composed of three parts:

- **One 1×1 standard convolution** is used to generate a general feature map.
- **Three 3×3 atrous convolutions** with different dilation rates are used to generate three feature maps at different scales. The dilation rates are set to 6, 12 and 18.
- **One Image Pooling** layer to obtain a global feature map.

The four feature maps are then concatenated and fed into a standard 1×1 convolution to generate the final semantic features f_4.

Decoder

The main purpose of the decoder network is to decode the obtained features by upsampling and convoluting to restore them to the original image size, resulting in the fire segmentation result. The implementation process of the decoder network is as follows:

1. Apply a standard 1×1 convolution with 256 convolution kernels to adjust the number of channels of features f_1, f_2 and f_3 to the ones of f_4.
2. Upsample features f_1, f_3, and f_4 to have the same feature size as f_2.
3. Concatenate these features and use a 3×3 convolution to re-adjust the channel number to 2, namely the background and foreground masks.
4. Upsample the last obtained spatial features to set the final segmentation images to the size of the original image.

In [13]:
```
from src.model.model import FireSegmentationModel

model = FireSegmentationModel(RESIZE_SHAPE, DEVICE)
```

In [14]:
```
# Set the batch size used for training.
TRAIN_BATCH_SIZE = 2
# Set the batch size used for evaluation.
EVAL_BATCH_SIZE = 2
```

In [15]:
```
from torchinfo import summary

summary(model, input_size=(TRAIN_BATCH_SIZE, 3, *RESIZE_SHAPE))
```

Out[15]:

```
==========================================================================================
====================
Layer (type:depth-idx)                                  Output Shape              Par
am #
==========================================================================================
====================
FireSegmentationModel                                   [2, 2, 512, 512]          --
├─Encoder: 1-1                                          [2, 16, 256, 256]         --
│    └─DCNN: 2-1                                        [2, 16, 256, 256]         --
│    │    └─Sequential: 3-1                             [2, 16, 256, 256]         464
│    │    └─Sequential: 3-2                             [2, 16, 256, 256]         752
│    │    └─Sequential: 3-3                             [2, 24, 128, 128]         7,8
80
│    │    └─Sequential: 3-4                             [2, 40, 64, 64]           56,
168
│    │    └─Sequential: 3-5                             [2, 160, 32, 32]          2,3
88,040
│    └─ASPP: 2-2                                        [2, 256, 32, 32]          --
│    │    └─Sequential: 3-6                             [2, 256, 32, 32]          41,
472
│    │    └─Sequential: 3-7                             [2, 256, 32, 32]          36
9,152
│    │    └─Sequential: 3-8                             [2, 256, 32, 32]          36
9,152
│    │    └─Sequential: 3-9                             [2, 256, 32, 32]          36
9,152
│    │    └─Sequential: 3-10                            [2, 256, 1, 1]            41,
472
│    │    └─Sequential: 3-11                            [2, 256, 32, 32]          32
8,192
├─Decoder: 1-2                                          [2, 2, 512, 512]          --
│    └─Sequential: 2-3                                  [2, 256, 256, 256]        --
│    │    └─Conv2d: 3-12                                [2, 256, 256, 256]        4,0
96
│    │    └─BatchNorm2d: 3-13                           [2, 256, 256, 256]        512
│    │    └─ReLU: 3-14                                  [2, 256, 256, 256]        --
│    └─Sequential: 2-4                                  [2, 256, 128, 128]        --
│    │    └─Conv2d: 3-15                                [2, 256, 128, 128]        6,1
44
│    │    └─BatchNorm2d: 3-16                           [2, 256, 128, 128]        512
│    │    └─ReLU: 3-17                                  [2, 256, 128, 128]        --
│    └─Sequential: 2-5                                  [2, 256, 64, 64]          --
│    │    └─Conv2d: 3-18                                [2, 256, 64, 64]          10,
240
│    │    └─BatchNorm2d: 3-19                           [2, 256, 64, 64]          512
│    │    └─ReLU: 3-20                                  [2, 256, 64, 64]          --
│    └─Upsample: 2-6                                    [2, 256, 128, 128]        --
│    └─Upsample: 2-7                                    [2, 256, 128, 128]        --
│    └─Upsample: 2-8                                    [2, 256, 128, 128]        --
│    └─Sequential: 2-9                                  [2, 2, 126, 126]          --
│    │    └─Conv2d: 3-21                                [2, 256, 126, 126]        2,3
59,296
│    │    └─BatchNorm2d: 3-22                           [2, 256, 126, 126]        512
│    │    └─ReLU: 3-23                                  [2, 256, 126, 126]        --
│    │    └─Conv2d: 3-24                                [2, 2, 126, 126]          514
│    └─Upsample: 2-10                                   [2, 2, 512, 512]          --
==========================================================================================
====================
Total params: 6,354,234
Trainable params: 6,354,234
```

```
Non-trainable params: 0
Total mult-adds (G): 82.29
================================================================================
====================
Input size (MB): 6.29
Forward/backward pass size (MB): 1762.76
Params size (MB): 25.42
Estimated Total Size (MB): 1794.47
================================================================================
====================
```

Training

Dataset

The train, validation and test dataloaders are obtained from the previously split train, validation and test images and masks.

The train dataloader uses a dataset where data augmentation is applied to the retrieved images and masks. In particular, the following operations are applied on both images and masks:

- Their perspective can be randomly distorted by a factor up to 0.3.
- They can be randomly horizontally flipped.
- A *random affine* tranformation can be finally applied to them composed of a rotation in the range $[-45^{\circ}, +45^{\circ}]$, a translation of fraction 0.1 of the pixels on the horizontal and vertical axis and a scaling between the factors $[0.5, 1.5]$.

A further transformation is applied to just the images of the train dataloader, which consist in a *color jitter* tranformation which can change the brightness of a factor between -0.2 and 0.2 and the hue of -0.05 and 0.05.

The images of both the train, test and validation dataloaders go all through an additional transformation, which consists in the standard scaling of their values applid to each channel separately. The means of each channel μ_B, μ_G and μ_R and their standard deviations σ_B, σ_G and σ_R are estimated on the train dataset.

The standardization of a channel C of an image x is computed as: $$x'_C = \frac{x_C - \mu_C}{\sigma_C}$$

In [16]:
```
import os
from numpy import save as np_save

train_mean = np.mean(X_train, axis=(-4, -3, -2))
train_std = np.std(X_train, axis=(-4, -3, -2))

train_mean_std_file_path = os.path.join(BASE_DIR, 'model', 'mean-std.npy')
os.makedirs(os.path.dirname(train_mean_std_file_path), exist_ok=True)
np_save(train_mean_std_file_path, (train_mean, train_std))
```

In [17]:
```
CHANNELS = ['B', 'G', 'R']

print('Mean of the training images per channel:',
      "; ".join([f'{c}: {m:.2f}' for c, m in zip(CHANNELS, train_mean)]))
print('Standard deviation of the training images per channel:',
      "; ".join([f'{c}: {m:.2f}' for c, m in zip(CHANNELS, train_std)]))
```

```
Mean of the training images per channel: B: 121.33; G: 119.34; R: 114.98
Standard deviation of the training images per channel: B: 69.79; G: 66.86; R: 65.85
```

In [18]:
```
from src.data.dataloaders import get_dataloader

train_loader = get_dataloader(
    X_train, y_train, train_mean, train_std, batch_size=TRAIN_BATCH_SIZE,
    shuffle=True, apply_augmentation=True)

val_loader = get_dataloader(
    X_val, y_val, train_mean, train_std, batch_size=EVAL_BATCH_SIZE,
    shuffle=False, apply_augmentation=False)

test_loader = get_dataloader(
    X_test, y_test, train_mean, train_std, batch_size=EVAL_BATCH_SIZE,
    shuffle=False, apply_augmentation=False)
```

Training Utilities

The training utilities and parameters are assigned.

- `Lion` is used as an optimizer with learning rate $1e-4$ and weight decay $1e-5$.
- Two learning rate schedulers are considered.
  - The first is a `StepLR` scheduler that simply decrements the learning rate of the optimizer of a factor 0.94 after every epoch.
  - The second is a `ReduceLROnPlateau` scheduler that reduces the learning rate step when a plateau is reached. It reduces the learning rate of 0.98 if the training *loss* does not decrease of a factor $1e-6$ after 300 batch steps.
- A checkpoint monitor is initialized to control the performances of the model on the validation set, by saving the best resulting weights which consist in the maximal accuracy computed as the sum of the *Mean Pixel Average (MPA)* and the *Mean Intersection over Union (MIoU)*.
- The number of *epochs* is set to 30.
- Validation is applied every other 200 batch steps and at the end of every epoch.

In [19]:
```
from torch.optim.lr_scheduler import StepLR, ReduceLROnPlateau
from src.training.lion import Lion

# Set the optimizer.
optimizer = Lion(model.parameters(), lr=1e-4, weight_decay=1e-5)

# Set the Learning Rate Schedulers.
step_lr_scheduler = StepLR(optimizer, gamma=.94, step_size=1)
plateau_lr_scheduler = ReduceLROnPlateau(
```

```
    optimizer, factor=.98, patience=300, threshold=1e-6)
lr_schedulers=[step_lr_scheduler, plateau_lr_scheduler]
```

In [20]:
```
import os
from src.training.utils import Checkpoint

# Set the number of epochs.
EPOCHS = 30

# Set the validation step.
VAL_STEP = 200

# Set the checkpoint monitor.
checkpoint = Checkpoint(os.path.join(BASE_DIR, 'model', 'checkpoints.pth'))
```

Training Loop

The training procedure is performed. The loss criterion is the *Focal Loss* between the logits of the predictions and the ground truth. This criterion is used in order for the model to focus its training on the harder sematic segmentation to obtain, namely the fire itself.

The *Mean Pixel Average (MPA)* and the *Mean Intersection over Union (MIoU)* are tracked as well on both the train and validation sets.

$$MPA = \frac{1}{k + 1}\sum\limits_{i = 0}^k \frac{p_{ii}}{\sum_{j=0}^k p_{ij}}$$$$MIoU = \frac{1}{k + 1}\sum\limits_{i = 0}^k \frac{p_{ii}}{\sum_{j=0}^k p_{ij} + \sum_{j=0}^k p_{ji} - p_{ii}} $$ where k represents the number of classes ($k = 1$, including fire and background), and p_{ij} represents the number of pixels that belong to class i but are predicted to be class j ; p_{ii} represents the number of pixels that belong to class i and are correctly predicted as class i.

⚠️ Warning for reproducibility

Code is not fully reproducible because the upsampling algorithms `upsample_bilinear2d` adopted in the *Decoder* module is not deterministic. This operation may produce nondeterministic gradients when given tensors on a CUDA device.

Different results may be obtained when re-running the pipeline.

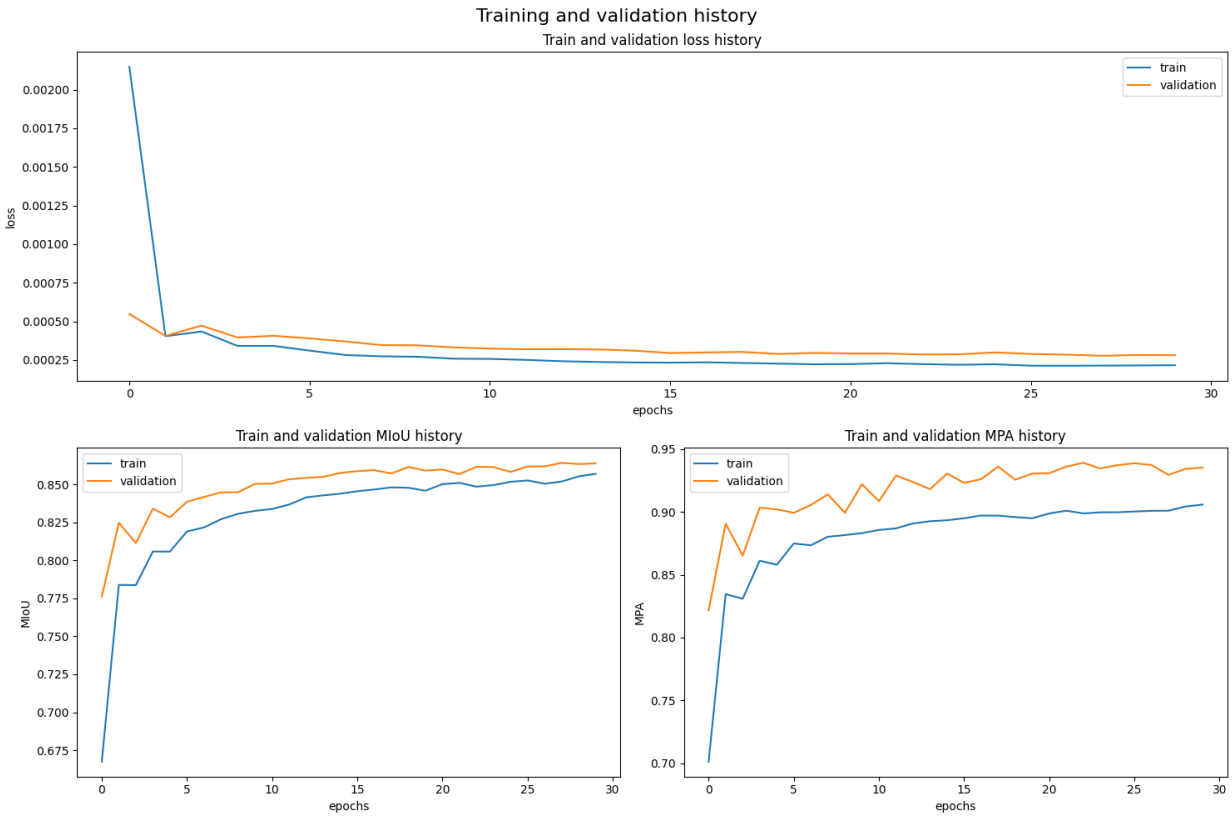
In [21]:
```
from src.training.training import train

history = train(model, optimizer, train_loader, val_loader, EPOCHS, VAL_STEP,
                DEVICE, checkpoint, lr_schedulers, reload_best_weights=True)
```

```
Epoch 1/30
[723/723] - 191s - train: { loss: 0.00215 - MPA: 70.1% - MiOU: 66.7% } - val: { loss:
0.000548 - MPA: 82.2% - MiOU: 77.6% } - lr: 0.0001
Epoch 2/30
[723/723] - 186s - train: { loss: 0.000404 - MPA: 83.5% - MiOU: 78.4% } - val: { los
s: 0.000406 - MPA: 89.1% - MiOU: 82.5% } - lr: 9.03e-05
Epoch 3/30
[723/723] - 186s - train: { loss: 0.000434 - MPA: 83.1% - MiOU: 78.4% } - val: { los
s: 0.000472 - MPA: 86.5% - MiOU: 81.1% } - lr: 8.15e-05
Epoch 4/30
[723/723] - 180s - train: { loss: 0.000341 - MPA: 86.1% - MiOU: 80.6% } - val: { los
s: 0.000395 - MPA: 90.3% - MiOU: 83.4% } - lr: 7.36e-05
Epoch 5/30
[723/723] - 192s - train: { loss: 0.000341 - MPA: 85.8% - MiOU: 80.6% } - val: { los
s: 0.000407 - MPA: 90.2% - MiOU: 82.8% } - lr: 6.64e-05
Epoch 6/30
[723/723] - 187s - train: { loss: 0.000311 - MPA: 87.5% - MiOU: 81.9% } - val: { los
s: 0.00039 - MPA: 89.9% - MiOU: 83.9% } - lr: 6e-05
Epoch 7/30
[723/723] - 185s - train: { loss: 0.000282 - MPA: 87.3% - MiOU: 82.2% } - val: { los
s: 0.00037 - MPA: 90.6% - MiOU: 84.2% } - lr: 5.31e-05
Epoch 8/30
[723/723] - 181s - train: { loss: 0.000274 - MPA: 88% - MiOU: 82.7% } - val: { loss:
0.000346 - MPA: 91.4% - MiOU: 84.5% } - lr: 4.79e-05
Epoch 9/30
[723/723] - 177s - train: { loss: 0.000271 - MPA: 88.2% - MiOU: 83.1% } - val: { los
s: 0.000345 - MPA: 89.9% - MiOU: 84.5% } - lr: 4.24e-05
Epoch 10/30
[723/723] - 177s - train: { loss: 0.000258 - MPA: 88.3% - MiOU: 83.3% } - val: { los
s: 0.000331 - MPA: 92.2% - MiOU: 85% } - lr: 3.83e-05
Epoch 11/30
[723/723] - 178s - train: { loss: 0.000257 - MPA: 88.6% - MiOU: 83.4% } - val: { los
s: 0.000324 - MPA: 90.8% - MiOU: 85.1% } - lr: 3.45e-05
Epoch 12/30
[723/723] - 179s - train: { loss: 0.00025 - MPA: 88.7% - MiOU: 83.7% } - val: { loss:
0.000319 - MPA: 92.9% - MiOU: 85.3% } - lr: 3.06e-05
Epoch 13/30
[723/723] - 177s - train: { loss: 0.000242 - MPA: 89.1% - MiOU: 84.1% } - val: { los
s: 0.00032 - MPA: 92.4% - MiOU: 85.4% } - lr: 2.76e-05
Epoch 14/30
[723/723] - 179s - train: { loss: 0.000237 - MPA: 89.3% - MiOU: 84.3% } - val: { los
s: 0.000318 - MPA: 91.8% - MiOU: 85.5% } - lr: 2.44e-05
Epoch 15/30
[723/723] - 179s - train: { loss: 0.000234 - MPA: 89.3% - MiOU: 84.4% } - val: { los
s: 0.00031 - MPA: 93.1% - MiOU: 85.8% } - lr: 2.2e-05
Epoch 16/30
[723/723] - 179s - train: { loss: 0.000233 - MPA: 89.5% - MiOU: 84.6% } - val: { los
s: 0.000295 - MPA: 92.3% - MiOU: 85.9% } - lr: 1.99e-05
Epoch 17/30
[723/723] - 178s - train: { loss: 0.000235 - MPA: 89.7% - MiOU: 84.7% } - val: { los
s: 0.000299 - MPA: 92.6% - MiOU: 85.9% } - lr: 1.8e-05
Epoch 18/30
[723/723] - 178s - train: { loss: 0.00023 - MPA: 89.7% - MiOU: 84.8% } - val: { loss:
0.000302 - MPA: 93.6% - MiOU: 85.7% } - lr: 1.62e-05
Epoch 19/30
[723/723] - 179s - train: { loss: 0.000226 - MPA: 89.6% - MiOU: 84.8% } - val: { los
s: 0.000289 - MPA: 92.6% - MiOU: 86.1% } - lr: 1.43e-05
Epoch 20/30
[723/723] - 194s - train: { loss: 0.000222 - MPA: 89.5% - MiOU: 84.6% } - val: { los
s: 0.000296 - MPA: 93.1% - MiOU: 85.9% } - lr: 1.29e-05
```

```
Epoch 21/30
[723/723] - 182s - train: { loss: 0.000224 - MPA: 89.9% - MiOU: 85% } - val: { loss:
0.000292 - MPA: 93.1% - MiOU: 86% } - lr: 1.15e-05
Epoch 22/30
[723/723] - 177s - train: { loss: 0.000229 - MPA: 90.1% - MiOU: 85.1% } - val: { los
s: 0.000292 - MPA: 93.6% - MiOU: 85.7% } - lr: 1.03e-05
Epoch 23/30
[723/723] - 178s - train: { loss: 0.000223 - MPA: 89.9% - MiOU: 84.9% } - val: { los
s: 0.000285 - MPA: 93.9% - MiOU: 86.2% } - lr: 9.34e-06
Epoch 24/30
[723/723] - 176s - train: { loss: 0.000219 - MPA: 90% - MiOU: 85% } - val: { loss: 0.
000287 - MPA: 93.5% - MiOU: 86.1% } - lr: 8.26e-06
Epoch 25/30
[723/723] - 177s - train: { loss: 0.000222 - MPA: 90% - MiOU: 85.2% } - val: { loss:
0.000299 - MPA: 93.7% - MiOU: 85.8% } - lr: 7.46e-06
Epoch 26/30
[723/723] - 177s - train: { loss: 0.000213 - MPA: 90% - MiOU: 85.3% } - val: { loss:
0.000289 - MPA: 93.9% - MiOU: 86.2% } - lr: 6.6e-06
Epoch 27/30
[723/723] - 178s - train: { loss: 0.000213 - MPA: 90.1% - MiOU: 85% } - val: { loss:
0.000284 - MPA: 93.7% - MiOU: 86.2% } - lr: 5.96e-06
Epoch 28/30
[723/723] - 177s - train: { loss: 0.000214 - MPA: 90.1% - MiOU: 85.2% } - val: { los
s: 0.000277 - MPA: 92.9% - MiOU: 86.4% } - lr: 5.38e-06
Epoch 29/30
[723/723] - 175s - train: { loss: 0.000215 - MPA: 90.4% - MiOU: 85.5% } - val: { los
s: 0.000282 - MPA: 93.4% - MiOU: 86.3% } - lr: 4.85e-06
Epoch 30/30
[723/723] - 178s - train: { loss: 0.000216 - MPA: 90.6% - MiOU: 85.7% } - val: { los
s: 0.000281 - MPA: 93.5% - MiOU: 86.4% } - lr: 4.38e-06
```

In [22]:

```
from src.training.analysis import plot_training_history

plot_training_history(history)
```



Results

In this section, the predictions of the model are evaluated and some samples of the outputed segmentation masks are shown.

In [23]:
```
checkpoint.load_best_weights(model)
model.eval();
```

Validation Results

In the original experiment no results were provided on the validation set, hereby we present ours without any comparison. The metrics used for evaluation are MPA, MIoU and FPS.

In [24]:
```
from src.training.training import validate

# Get the validation evaluation results.
val_loss, val_mpa, val_miou, val_fps = validate(
    model, val_loader, DEVICE, resize_evaluation_shape=ORIGINAL_SHAPE)
```

In [25]:
```
print('Validation loss:', f'{val_loss:.3g}')
print('Validation MPA:', f'{val_mpa * 100:.3g}')
print('Validation MIoU:', f'{val_miou * 100:.3g}')
print('Validation FPS:', f'{val_fps:.3g}')
```

```
Validation loss: 0.000285
Validation MPA: 93.9
Validation MIoU: 86.2
Validation FPS: 67.5
```

Test Results

The results of the model are evaluated on the test set and compared to the outcomes of the original paper. The metrics used for evaluation are once again MPA, MIoU and FPS.

	Original Paper	Our results
MPA (%)	92.46	94.3
MIoU (%)	86.98	86.1
FPS	59	58.5

In [26]:
```
from src.training.training import validate

# Get the test evaluation results.
test_loss, test_mpa, test_miou, test_fps = validate(
    model, test_loader, DEVICE, resize_evaluation_shape=ORIGINAL_SHAPE)
```

In [27]:
```
print('Test loss:', f'{test_loss:.3g}')
print('Test MPA:', f'{test_mpa * 100:.3g}')
print('Test MIoU:', f'{test_miou * 100:.3g}')
print('Test FPS:', f'{test_fps:.3g}')
```

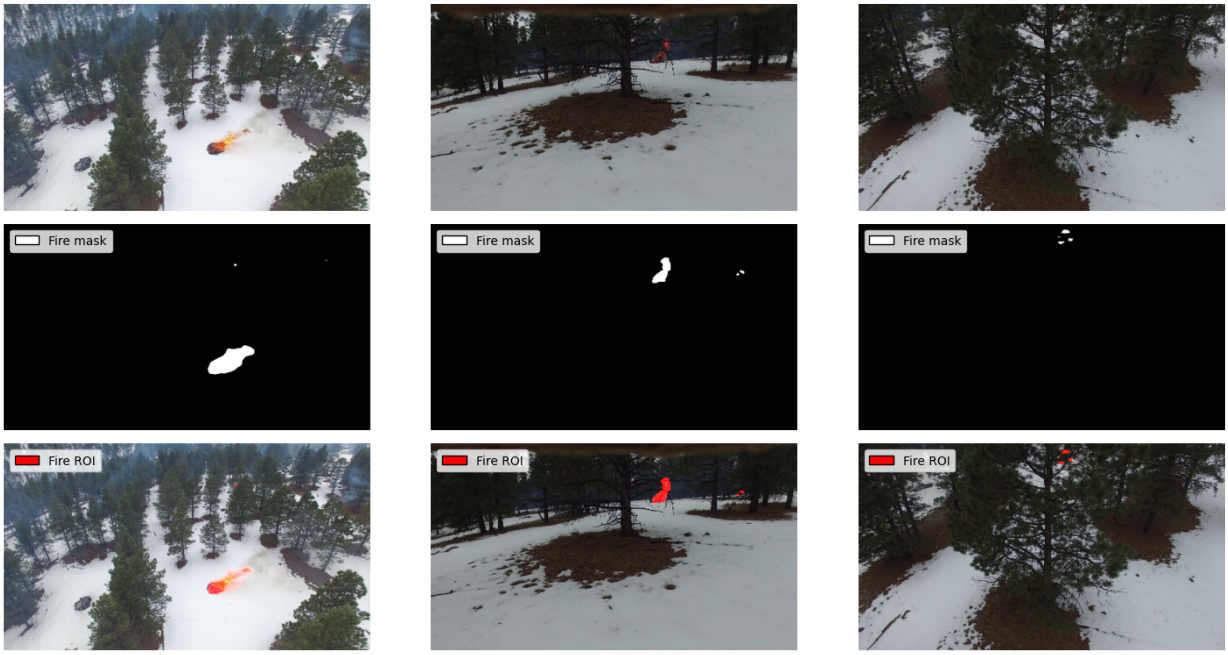
```
Test loss: 0.000288
Test MPA: 93.9
Test MIoU: 85.9
Test FPS: 66.9
```

Qualitative Analysis

In [28]:
```
from src.prediction.predict import predict
from src.data.dataset_handler import resize_images

# Get the model test predictions to the original size.
y_pred = predict(model, test_loader, DEVICE)
```

In [29]:
```
from src.data.analysis import plot_dataset_samples

plot_dataset_samples(
    X_test, y_pred,
    title='Representation of 3 images from the dataset '
    'along with their predicted fire mask and the highlighted segmentation',
    resize_shape=ORIGINAL_SHAPE)
```

Representation of 3 images from the dataset along with their predicted fire mask and the highlighted segmentation



Data Visualizations

In [107…

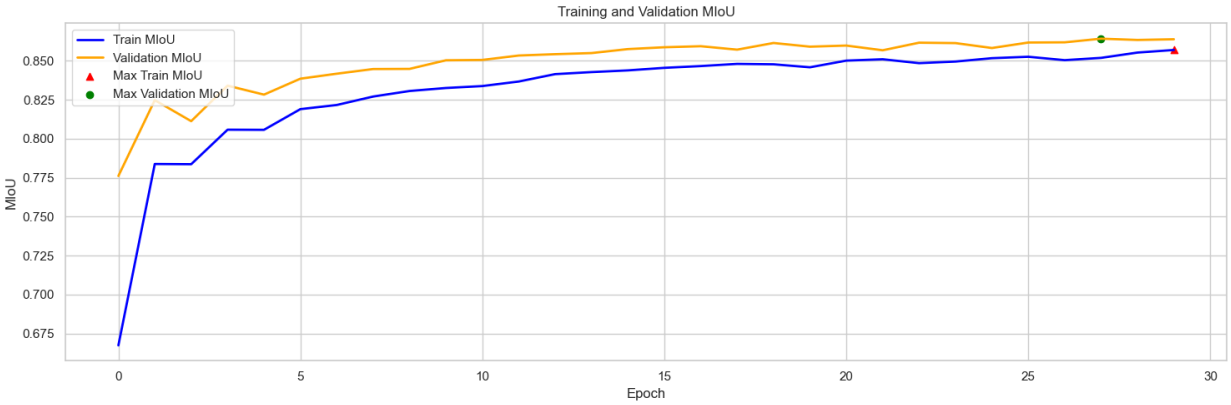
```
import keras
from tabulate import tabulate
import matplotlib.pyplot as plt

print("\n")
print(tabulate([['Train', {X_train.shape}, {y_train.shape}], ['Validation',{X_val.shap
               headers=['Name','X.shape', 'y.shape']))

#-------------------------------------------------------------------------------------
print("\n")
print(tabulate([['Train', "; ".join([f'{c}: {m:.2f}' for c, m in zip(CHANNELS, train_m
               headers=['Name','Mean of the training images per channel:', 'Standard d
print("\n")
#-------------------------------------------------------------------------------------

print("\n")


import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Assuming 'history' is a dictionary containing your training history

# Function to find the index of the maximum value in a list
def find_max_index(data):
    return np.argmax(data)

# Plotting Miou and MPA
plt.figure(figsize=(15, 10))

# Set a seaborn style for better aesthetics
sns.set(style="whitegrid")
```

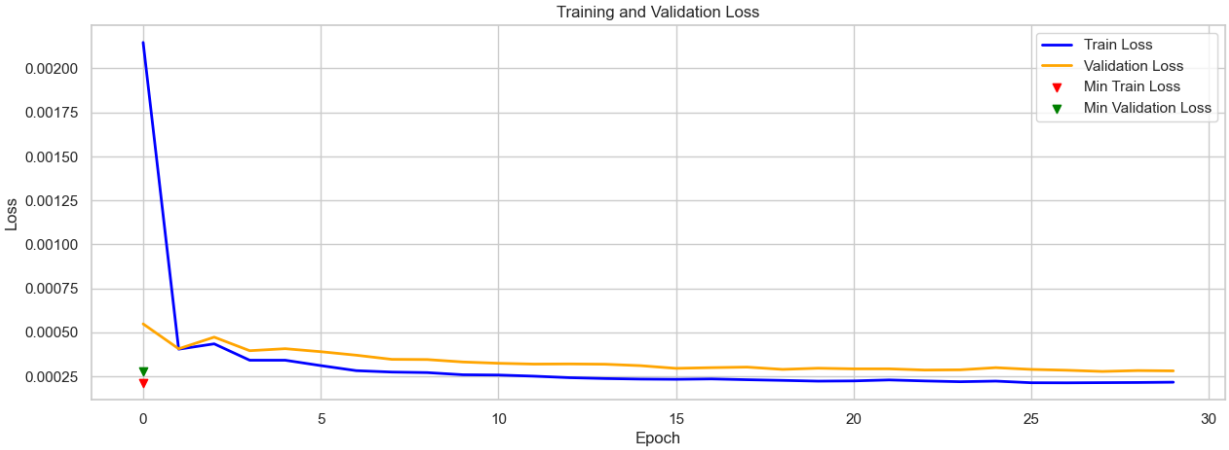
```
# Miou
plt.subplot(2, 1, 1)
plt.plot(history['train_miou'], label='Train MIoU', color='blue', linewidth=2)
plt.plot(history['val_miou'], label='Validation MIoU', color='orange', linewidth=2)

max_train_miou_index = find_max_index(history['train_miou'])
max_val_miou_index = find_max_index(history['val_miou'])

plt.scatter(max_train_miou_index, max(history['train_miou']), marker = '^', color='red
plt.scatter(max_val_miou_index, max(history['val_miou']), color='green', label='Max Va
plt.legend(loc='upper left')
plt.title('Training and Validation MIoU')
plt.xlabel('Epoch')
plt.ylabel('MIoU')
plt.grid(True)

# MPA
plt.subplot(2, 1, 2)
plt.plot(history['train_mpa'], label='Train MPA', color='blue', linewidth=2)
plt.plot(history['val_mpa'], label='Validation MPA', color='orange', linewidth=2)

max_train_mpa_index = find_max_index(history['train_mpa'])
max_val_mpa_index = find_max_index(history['val_mpa'])

plt.scatter(max_train_mpa_index, max(history['train_mpa']), color='red', marker='^', l
plt.scatter(max_val_mpa_index, max(history['val_mpa']), color='green', label='Max Vali
plt.legend(loc='upper left')
plt.title('Training and Validation MPA')
plt.xlabel('Epoch')
plt.ylabel('MPA')
plt.grid(True)

plt.tight_layout()
plt.show()

# Plotting Loss
plt.figure(figsize=(15, 5))

# Loss
plt.plot(history['train_loss'], label='Train Loss', color='blue', linewidth=2)
plt.plot(history['val_loss'], label='Validation Loss', color='orange', linewidth=2)

min_train_loss_index = find_max_index(history['train_loss'])
min_val_loss_index = find_max_index(history['val_loss'])

plt.scatter(min_train_loss_index, min(history['train_loss']), color='red', marker='v',
plt.scatter(min_val_loss_index, min(history['val_loss']), color='green', marker='v', l
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)

plt.show()


#-------------------------------------------------------------------------------
print("\n")
print('Validation loss:', f'{val_loss:.3g}')
print('Validation MPA:', f'{val_mpa * 100:.3g}')
```

```
print('Validation MIoU:', f'{val_miou * 100:.3g}')
print('Validation FPS:', f'{val_fps:.3g}')
print("\n")
#-----------------------------------------------------------------------------------------
print('Test loss:', f'{test_loss:.3g}')
print('Test MPA:', f'{test_mpa * 100:.3g}')
print('Test MIoU:', f'{test_miou * 100:.3g}')
print('Test FPS:', f'{test_fps:.3g}')
print("\n")
#-----------------------------------------------------------------------------------------
y_pred = predict(model, test_loader, DEVICE)
print("\n")
#-----------------------------------------------------------------------------------------
summary(model.cuda(), input_size=(TRAIN_BATCH_SIZE, 3, *RESIZE_SHAPE), col_names=["ker
```

```
Name        X.shape                 y.shape
----------  ----------------------  ------------------
Train       {(1446, 512, 512, 3)}  {(1446, 512, 512)}
Validation  {(256, 512, 512, 3)}   {(256, 512, 512)}
Test        {(301, 512, 512, 3)}   {(301, 512, 512)}
```

```
Name    Mean of the training images per channel:    Standard deviation of the trainin
g images per channel:
------  ------------------------------------------  ------------------------------
----------------------
Train   B: 121.33; G: 119.34; R: 114.98             B: 69.79; G: 66.86; R: 65.85
```






```
Validation loss: 0.000285
Validation MPA: 93.9
Validation MIoU: 86.2
Validation FPS: 67.5


Test loss: 0.000288
Test MPA: 93.9
Test MIoU: 85.9
Test FPS: 66.9
```

Out[107]:

```
==========================================================================================
==========================================================================================
===============
Layer (type (var_name))                                  Kernel Shape
Input Shape               Output Shape              Param #                   Mult-Ad
ds
==========================================================================================
==========================================================================================
===============
FireSegmentationModel (FireSegmentationModel)            --
[2, 3, 512, 512]          [2, 2, 512, 512]          --                        --
├─Encoder (encoder)                                      --
[2, 3, 512, 512]          [2, 16, 256, 256]         --                        --
│    └─DCNN (dcnn)                                       --
[2, 3, 512, 512]          [2, 16, 256, 256]         --                        --
│    │    └─Sequential (input_convolution)               --
[2, 3, 512, 512]          [2, 16, 256, 256]         464                       56,623,
168
│    │    └─Sequential (bottlenecks_sequential_1)        --
[2, 16, 256, 256]         [2, 16, 256, 256]         752                       85,983,
424
│    │    └─Sequential (bottlenecks_sequential_2)        --
[2, 16, 256, 256]         [2, 24, 128, 128]         7,880                     337,90
4,896
│    │    └─Sequential (bottlenecks_sequential_3)        --
[2, 24, 128, 128]         [2, 40, 64, 64]           56,168                    301,44
9,184
│    │    └─Sequential (bottlenecks_sequential_4)        --
[2, 40, 64, 64]           [2, 160, 32, 32]          2,388,040                 2,736,8
65,664
│    └─ASPP (aspp)                                       --
[2, 160, 32, 32]          [2, 256, 32, 32]          --                        --
│    │    └─Sequential (standard_convolution)            --
[2, 160, 32, 32]          [2, 256, 32, 32]          41,472                    83,887,
104
│    │    └─Sequential (atrous_convolution_1)            --
[2, 160, 32, 32]          [2, 256, 32, 32]          369,152                   754,97
5,744
│    │    └─Sequential (atrous_convolution_2)            --
[2, 160, 32, 32]          [2, 256, 32, 32]          369,152                   754,97
5,744
│    │    └─Sequential (atrous_convolution_3)            --
[2, 160, 32, 32]          [2, 256, 32, 32]          369,152                   754,97
5,744
│    │    └─Sequential (global_average_pooling)          --
[2, 160, 32, 32]          [2, 256, 1, 1]            41,472                    82,944
│    │    └─Sequential (final_convolution)               --
[2, 1280, 32, 32]         [2, 256, 32, 32]          328,192                   671,08
9,664
├─Decoder (decoder)                                      --
[2, 16, 256, 256]         [2, 2, 512, 512]          --                        --
│    └─Sequential (convolution_f1)                       --
[2, 16, 256, 256]         [2, 256, 256, 256]        --                        --
│    │    └─Conv2d (0)                                   [1, 1]
[2, 16, 256, 256]         [2, 256, 256, 256]        4,096                     536,87
0,912
│    │    └─BatchNorm2d (1)                              --
[2, 256, 256, 256]        [2, 256, 256, 256]        512                       1,024
│    │    └─ReLU (2)                                     --
[2, 256, 256, 256]        [2, 256, 256, 256]        --                        --
```

```
│    └─Sequential (convolution_f2)                        --
[2, 24, 128, 128]         [2, 256, 128, 128]        --                        --
│    │    └─Conv2d (0)                                   [1, 1]
[2, 24, 128, 128]         [2, 256, 128, 128]        6,144                     201,32
6,592
│    │    └─BatchNorm2d (1)                              --
[2, 256, 128, 128]        [2, 256, 128, 128]        512                       1,024
│    │    └─ReLU (2)                                     --
[2, 256, 128, 128]        [2, 256, 128, 128]        --                        --
│    └─Sequential (convolution_f3)                        --
[2, 40, 64, 64]           [2, 256, 64, 64]          --                        --
│    │    └─Conv2d (0)                                   [1, 1]
[2, 40, 64, 64]           [2, 256, 64, 64]          10,240                    83,886,
080
│    │    └─BatchNorm2d (1)                              --
[2, 256, 64, 64]          [2, 256, 64, 64]          512                       1,024
│    │    └─ReLU (2)                                     --
[2, 256, 64, 64]          [2, 256, 64, 64]          --                        --
│    └─Upsample (upsample_f1)                             --
[2, 256, 256, 256]        [2, 256, 128, 128]        --                        --
│    └─Upsample (upsample_f3)                             --
[2, 256, 64, 64]          [2, 256, 128, 128]        --                        --
│    └─Upsample (upsample_f4)                             --
[2, 256, 32, 32]          [2, 256, 128, 128]        --                        --
│    └─Sequential (final_convolution)                     --
[2, 1024, 128, 128]       [2, 2, 126, 126]          --                        --
│    │    └─Conv2d (0)                                   [3, 3]
[2, 1024, 128, 128]       [2, 256, 126, 126]        2,359,296                 74,912,
366,592
│    │    └─BatchNorm2d (1)                              --
[2, 256, 126, 126]        [2, 256, 126, 126]        512                       1,024
│    │    └─ReLU (2)                                     --
[2, 256, 126, 126]        [2, 256, 126, 126]        --                        --
│    │    └─Conv2d (3)                                   [1, 1]
[2, 256, 126, 126]        [2, 2, 126, 126]          514                       16,320,
528
│    └─Upsample (final_upsample)                          --
[2, 2, 126, 126]          [2, 2, 512, 512]          --                        --
==========================================================================================
==========================================================================================
===============
Total params: 6,354,234
Trainable params: 6,354,234
Non-trainable params: 0
Total mult-adds (G): 82.29
==========================================================================================
==========================================================================================
===============
Input size (MB): 6.29
Forward/backward pass size (MB): 1762.76
Params size (MB): 25.42
Estimated Total Size (MB): 1794.47
==========================================================================================
==========================================================================================
===============
```

The Kernel crashed while executing code in the the current cell or a previous cell. Please review the code in the cell(s) to identify a possible cause of the failure. Click here for more info. View Jupyter log for further details.