Assignment 2
Submitted by: Adarsh Ghimire
Student ID: 100058927

General summary of the assignment.

In this assignment, the UNET architecture model is implemented, trained, and tested for the ISPRS Dataset segmentation task. The dataset comprises of 2400 images of size 300x300 in total. So, for the completeness and robustness of the model, the total data is divided into 3 sets:

1. Training Set of 2000 images
2. Validation Set of 200 images
3. Test Set of 200 images

The validation set has been chosen to find the optimal model parameters, and test set has been kept untouched until the model in finalized so that there is no information leakage.

The model has been trained in Kaggle for 20 epochs, with batch size of 10 only, and all the model parameters and optimizer states has been saved in regular manner to allow further training, if required. In addition, the best performing model on validation set is also saved so that it can later be used for inference purpose. Below is the summarized result of the best model:

Dataset	Loss (cross entropy)	Accuracy
Training set	0.59	80.033
Validation set	0.63	79.312
Test set	0.54	79.992

The models are large thus are saved in google drive for sharing. Please find the model from the below link.

Model folder link :
https://drive.google.com/drive/folders/13MimYs-8wyylYbM_q_kEvb4cmjGhPrqF?usp=sharing

Comprises of two files:

1. model_best.pth.tar → Is the best model
2. Checkpoint.pth.tar → is the current model state

Detailed explanation of the assignment

1. Initialization

Here the initial some parameters are initialized at beginning.

```
In [6]:
# Parameters
IN_CHANNELS =  3                            # Number of input channels (e.g. RGB)
MAIN_FOLDER = "../input/segmentation/patches/"   # Your "/path/to/the/Images/folder/"
BATCH_SIZE = 10             # Number of samples in a mini-batch, example 10
LABELS = ["roads", "buildings", "low veg.", "trees", "cars", "clutter"] # Label names
N_CLASSES = len(LABELS)                     # Number of classes
weights = torch.ones(N_CLASSES)             # Weights for class balancing
DATA_FOLDER = MAIN_FOLDER + 'Images/Image_{}.tif'
LABELS_FOLDER = MAIN_FOLDER + 'Labels/Label_{}.tif'
```

2. Some necessary functions

```
# Let's define the standard ISPRS color palette
palette = {0 : (255, 255, 255), # Impervious surfaces (white)
           1 : (0, 0, 255),     # Buildings (blue)
           2 : (0, 255, 255),   # Low vegetation (cyan)
           3 : (0, 255, 0),     # Trees (green)
           4 : (255, 255, 0),   # Cars (yellow)
           5 : (255, 0, 0),     # Clutter (red)
           6 : (0, 0, 0)}       # Undefined (black)
invert_palette = {v: k for k, v in palette.items()}
def convert_from_color(arr_3d, palette=invert_palette):
    """ RGB-color encoding to grayscale labels """ '(From 0 to 6)'
    arr_2d = np.zeros((arr_3d.shape[0], arr_3d.shape[1]), dtype=np.uint8)
    for c, i in palette.items():
        m = np.all(arr_3d == np.array(c).reshape(1, 1, 3), axis=2)
        arr_2d[m] = i
    return arr_2d
```

This part of the code is responsible for conversion of the RGB color encoding of the labels of the data to categorical value from 0 to 6.

```
class Load_dataset(torch.utils.data.Dataset):
    def __init__(self, ids):
        super(Load_dataset, self).__init__()
        # List of files
        self.data_files = [DATA_FOLDER.format(id) for id in ids]
        self.label_files = [LABELS_FOLDER.format(id) for id in ids]
        # Sanity check : raise an error if some files do not exist
        for f in self.data_files + self.label_files:
            if not os.path.isfile(f):
                raise KeyError('{} is not a file !'.format(f))
    def __len__(self):
        return len(self.data_files) # the length of the used data

    def __getitem__(self, idx):
#         Pre-processing steps
        #     # Data is normalized in [0, 1]
        self.data = 1/255 * np.asarray(io.imread(self.data_files[idx]).transpose((2,0,1)), dtype
='float32')
        self.label = np.asarray(convert_from_color(io.imread(self.label_files[idx])), dtype='int
64')
        data_p, label_p = self.data,  self.label
        # Return the torch.Tensor values
        return (torch.from_numpy(data_p),
                torch.from_numpy(label_p))
```

This is the custom dataset class for the custom ISPRS dataset so that later pytorch data loader can use it.

```python
def CrossEntropy2d(input, target, weight=None, size_average=True):
    """ 2D version of the cross entropy loss """
    dim = input.dim()
    if dim == 2:
        return F.cross_entropy(input, target, weight, size_average)
    elif dim == 4:
        output = input.view(input.size(0), input.size(1), -1)
        output = torch.transpose(output, 1, 2).contiguous()
        output = output.view(-1, output.size(2))
        target = target.view(-1)
        return F.cross_entropy(output, target, weight, size_average)
    else:
        raise ValueError('Expected 2 or 4 dimensions (got {})'.format(dim))
```

This function computes the cross-entropy loss for 2D data. Since the model outputs *(batch, classes_score, input_widht, input_height)* as the output, and the actual label is just (*batch, class_number, input_width, input_height*). So, the above function computes the loss among them.

```python
def metrics(predictions, gts, label_values=LABELS):
    cm = confusion_matrix(
        gts,
        predictions,
        range(len(label_values)))
    print("Confusion matrix :")
    print(cm)
    print("---")
    # Compute global accuracy
    total = sum(sum(cm))
    accuracy = sum([cm[x][x] for x in range(len(cm))])
    accuracy *= 100 / float(total)
    print("{} pixels processed".format(total))
    print("Total accuracy : {}%".format(accuracy))
    return accuracy
```

The function computes the confusion matrix and accuracy on per pixel basis.

3. Selecting training, validation, and test data

```
In [8]:
train_ids =list(range(0, 2000))
val_ids = list(range(2000,2200))
test_ids =  list(range(2200,2400))
```

```
In [9]:
trainset = Load_dataset(train_ids)
validationset = Load_dataset(val_ids)
testset = Load_dataset(test_ids)
```

```
In [10]:
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE, shuffle=True)
valloader = torch.utils.data.DataLoader(validationset, batch_size=BATCH_SIZE)
testloader = torch.utils.data.DataLoader(testset, batch_size=BATCH_SIZE)
```

4. Implementation of U-Net Model

The implementation of U-Net model has been modularized for easy and faster implementation. And, to avoid repetitive coding.

First, is the DoubleConv class which performs two convolution operations followed by the Relu activation. In between the convolution and activation operation batch normalization layer has been added for making learning efficient and to avoid overfitting. As it can be seen from summarized result above that the model is able to generalize.

```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),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

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

Second is the Down class which does maxpooling and then performs DoubleConv operation as described above.

```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)
```

Third is the Up class which performs the upscaling operation since we need to convert our encoded information back to original size. In this class, the upscaling can be performed in two ways has been implemented. First is using nn.Upsample for upscaling and another is using nn.ConvTranspose2d. Both works fine. After the upscaling operation feature map concatenation with feature map from down operation is performed followed by Double Convolution operation.

```python
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', align_corners=True)
            self.conv = DoubleConv(in_channels, out_channels, in_channels // 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])
        x = torch.cat([x2, x1], dim=1)
        return self.conv(x)
```

Below, is the overall architecture for the Unet.

```
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
```

5. Data Visualization

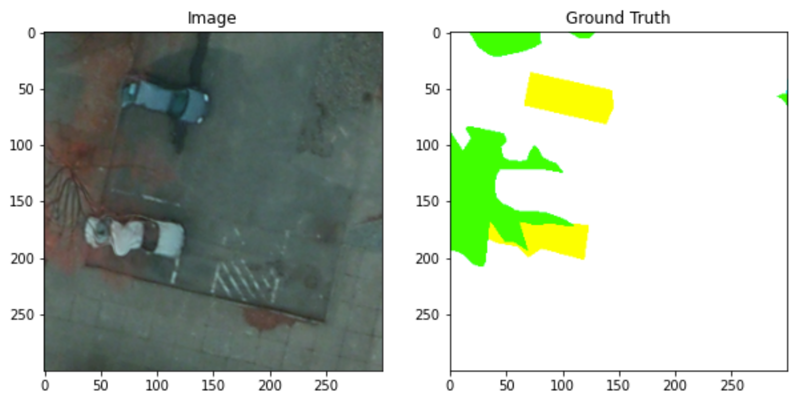
```
def convert_to_color(arr_2d, palette=palette):
    """ Encoding to RGB-color """
    n_channels = 3
    arr_3d = np.zeros((n_channels, arr_2d.shape[0], arr_2d.shape[1]))
    for c, i in palette.items():
#          print(arr_3d[:, c == arr_2d])
        arr_3d[0, arr_2d == c] = i[0]
        arr_3d[1, arr_2d == c] = i[1]
        arr_3d[2, arr_2d == c] = i[2]
    return arr_3d
```

This function will convert the class labels to corresponding color palette so that visualization can be done.

```
def show_image_and_label(image, label, pred_label=None):
    if pred_label is None:
        rows = 1
        columns = 2
        fig = plt.figure(figsize=(10, 10))
        fig.add_subplot(rows, columns, 1)
        plt.title("Image")
        plt.imshow(np.asarray(image).transpose(1,2,0))
        fig.add_subplot(rows, columns, 2)
        plt.imshow(label.transpose(1,2,0))
        plt.title("Ground Truth")
    else:
        rows = 1
        columns = 3
        fig = plt.figure(figsize=(10, 15))
        fig.add_subplot(rows, columns, 1)
        plt.imshow(np.asarray(image).transpose(1,2,0))
        plt.title("Image")
        fig.add_subplot(rows, columns, 2)
        plt.imshow(label.transpose(1,2,0))
        plt.title("Ground Truth")
        fig.add_subplot(rows, columns, 3)
        plt.imshow(pred_label.transpose(1,2,0))
        plt.title("Predicted Label")
```

This function plots the image, its ground truth and the prediction from the model if provided.
Example of image and its corresponding label.

In [26]:
```
show_image_and_label(images[image_number], label_image)
```



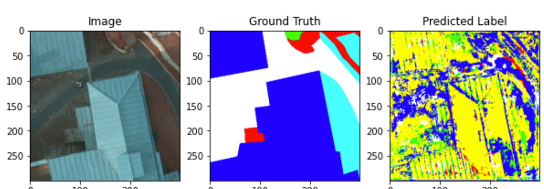
Visualizing the untrained model output, just to confirm that the process will work fine during training process. Checking the current untrained model loss, accuracy, and some visualization.
Randomly initialized untrained model summary :

Loss : 1.8572

Accuracy : 12.83%

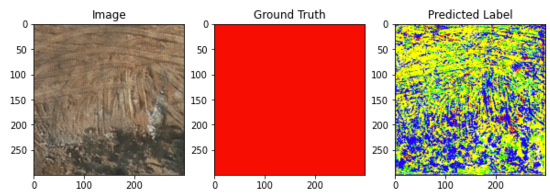
```
# Since the data are usually in GPU so they need to be detached before we can plot them

# Some Random prediction data sample 1
image_number = 0
print("Prediction from model of accuracy %0.2f"%random_acc)
color_label = convert_to_color(labels[image_number].cpu().detach().numpy())
pred_color_label = convert_to_color(np.argmax(output_pred_cpu, axis=1)[image_number])
show_image_and_label(images[image_number].cpu().detach().numpy(), color_label, pred_color_label)
```

```
Prediction from model of accuracy 12.84
```



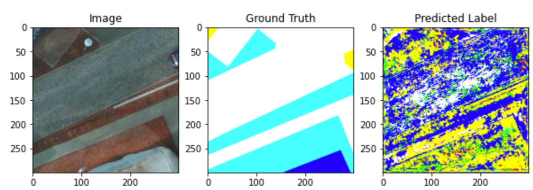
```
# Some Random prediction data sample 2
image_number = 1
print("Prediction from model of accuracy %0.2f"%random_acc)
color_label = convert_to_color(labels[image_number].cpu().detach().numpy())
pred_color_label = convert_to_color(np.argmax(output_pred_cpu, axis=1)[image_number])
show_image_and_label(images[image_number].cpu().detach().numpy(), color_label, pred_color_label)
```

```
Prediction from model of accuracy 12.84
```




```
# Some Random prediction data sample 3
image_number = 2
print("Prediction from model of accuracy %0.2f"%random_acc)
color_label = convert_to_color(labels[image_number].cpu().detach().numpy())
pred_color_label = convert_to_color(np.argmax(output_pred_cpu, axis=1)[image_number])
show_image_and_label(images[image_number].cpu().detach().numpy(), color_label, pred_color_label)
```

```
Prediction from model of accuracy 12.84
```



6. Training

```
# Redefing the metrics by commenting the printing of confusion matrix
# SO that it does not keep on printing during training
def metrics(predictions, gts, label_values=LABELS):
    cm = confusion_matrix(gts, predictions, range(len(label_values)))
    # Compute global accuracy
    total = sum(sum(cm))
    accuracy = sum([cm[x][x] for x in range(len(cm))])
    accuracy *= 100 / float(total)

    return accuracy
```

This function for calculating accuracy has been redefined so that the function do not keep on printing on every step of training

```
def save_checkpoint(state, is_best, filename='checkpoint.pth.tar'):
    torch.save(state, filename)
    if is_best:
        print("Saving best model !")
        shutil.copyfile(filename, 'model_best.pth.tar')
```

This function will save the model in regular manner as well as save the best performing model.

```
epochs = 20
learning_rate = 0.001
n_train = len(trainloader)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Using {} ".format(device))
# Change here to adapt to your data
# n_channels=3 for RGB images
# n_classes is the number of probabilities you want to get per pixel
net = UNet(n_channels=3, n_classes=6, bilinear=True)
```

```
Using cuda
```

Initial parameters setup for training, and Unet model object instantiation.

Training step

```
In [52]:
net.to(device=device)
optimizer = optim.RMSprop(net.parameters(), lr=learning_rate, weight_decay=1e-8, momentum=0.9)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'max', patience=2)
criterion = nn.CrossEntropyLoss()
print_every = 100

# 5. Begin training
train_loss_array = []
val_loss_array = []
val_acc_array = []
best_acc1 = 0
for epoch in range(epochs):
    net.train()
    train_loss = 0
    i = 0
    for batch in trainloader:
        i += 1
        optimizer.zero_grad()
        images = batch[0]
        true_masks = batch[1]

        assert images.shape[1] == net.n_channels, f'Network has been defined with {net.n_channel
s} input channels, ' f'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)


        masks_pred = net(images)

        loss = CrossEntropy2d(masks_pred, true_masks)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
```

```
        train_loss += loss.item()

        # Testing the model performance on Validation set on every 100 loop through training batch
        if i % print_every == 0:
            train_loss_array.append(train_loss/print_every)
            masks_pred_cpu = masks_pred.cpu().detach().numpy()
            true_masks_cpu = true_masks.cpu().detach().numpy()
            print('[%d epoch, %5d batch] Train loss: %.3f Train accuracy : %.3f' % (epoch + 1,
i, train_loss / print_every, metrics(np.argmax(masks_pred_cpu, axis=1).flatten(), true_masks_cp
u.flatten(), label_values=LABELS)))
            train_loss = 0.0

            # Validation data
            val_loss = 0.0
            val_accuracy = 0.0
            n_val = len(valloader)
            with torch.no_grad():
                net.eval()
                for data in valloader:
                    images = data[0]
                    true_masks = data[1]
                    images = images.to(device=device, dtype=torch.float32)
                    true_masks = true_masks.to(device=device)

                    # Making prediction
                    masks_pred = net(images)
                    val_batch_loss = CrossEntropy2d(masks_pred, true_masks)
                    val_loss += val_batch_loss.item()

                    masks_pred_cpu = masks_pred.cpu().detach().numpy()
                    true_masks_cpu = true_masks.cpu().detach().numpy()
                    val_accuracy += metrics(np.argmax(masks_pred_cpu, axis=1).flatten(), true_ma
sks_cpu.flatten(), label_values=LABELS)
                val_loss_array.append(val_loss/n_val)
                val_acc_array.append(val_accuracy/n_val)
                print("Val loss: %.3f Val accuracy : %.3f"%(val_loss/n_val, val_accuracy/n_val))
```

```
                print("Val loss: %.3f Val accuracy : %.3f"%(val_loss/n_val, val_accuracy/n_val))

                acc1 = val_accuracy/n_val
                is_best = acc1 > best_acc1
                best_acc1 = max(acc1, best_acc1)
                save_checkpoint({
                    'epoch': epoch + 1,
                    'state_dict': net.state_dict(),
                    'best_acc1': best_acc1,
                    'optimizer' : optimizer.state_dict(),
                }, is_best)
                print("******************************************")
            net.train()


print("Finish Training")
```

Above code snippet shows the training step followed by validation step for verification of the trained model. The model verification is done every 100th step. And if the validation model performance is better than other previous performance then the optimal model parameters are saved as *model.pt.tar*. And latest model trained is saved as *checkpoint.pt.tar* in the home directory. So, the checkpoint is saved to allow further training later, and best model is saved for best inferencing model.

Output of the above step.

```
[1 epoch,   100 batch] Train loss: 1.284 Train accuracy : 54.413
Val loss: 1.199 Val accuracy : 44.073
Saving best model !
******************************************
[1 epoch,   200 batch] Train loss: 1.156 Train accuracy : 51.017
Val loss: 1.202 Val accuracy : 50.357
Saving best model !
******************************************
[2 epoch,   100 batch] Train loss: 1.091 Train accuracy : 55.898
Val loss: 1.298 Val accuracy : 39.358
******************************************
[2 epoch,   200 batch] Train loss: 1.054 Train accuracy : 48.535
Val loss: 0.981 Val accuracy : 61.646
Saving best model !
******************************************
[3 epoch,   100 batch] Train loss: 1.010 Train accuracy : 58.542
Val loss: 0.948 Val accuracy : 65.042
Saving best model !
******************************************
[3 epoch,   200 batch] Train loss: 1.021 Train accuracy : 65.487
Val loss: 0.933 Val accuracy : 68.053
Saving best model !
******************************************
[4 epoch,   100 batch] Train loss: 0.984 Train accuracy : 53.837
Val loss: 0.905 Val accuracy : 64.908
******************************************
[4 epoch,   200 batch] Train loss: 0.954 Train accuracy : 69.153
Val loss: 0.891 Val accuracy : 66.716
******************************************
[5 epoch,   100 batch] Train loss: 0.943 Train accuracy : 63.159
Val loss: 1.195 Val accuracy : 58.726
******************************************
[5 epoch,   200 batch] Train loss: 0.923 Train accuracy : 64.167
Val loss: 0.855 Val accuracy : 68.756
Saving best model !
******************************************
```

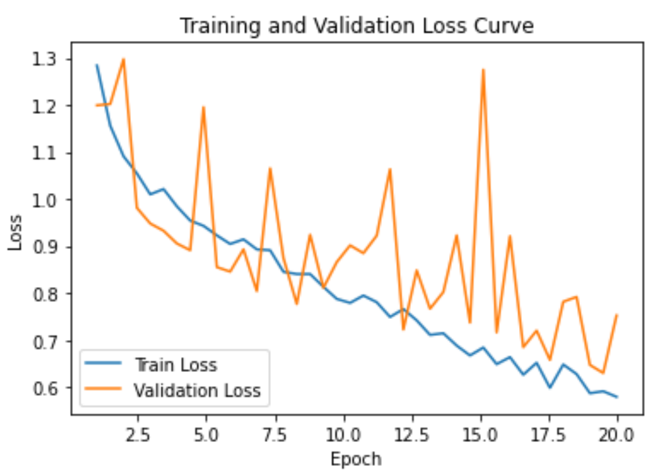
……………..

```
******************************************
[16 epoch,   100 batch] Train loss: 0.649 Train accuracy : 80.445
Val loss: 0.716 Val accuracy : 74.817
******************************************
[16 epoch,   200 batch] Train loss: 0.664 Train accuracy : 85.006
Val loss: 0.921 Val accuracy : 70.335
******************************************
[17 epoch,   100 batch] Train loss: 0.626 Train accuracy : 79.205
Val loss: 0.685 Val accuracy : 77.623
Saving best model !
******************************************
[17 epoch,   200 batch] Train loss: 0.652 Train accuracy : 81.420
Val loss: 0.720 Val accuracy : 76.652
******************************************
[18 epoch,   100 batch] Train loss: 0.598 Train accuracy : 84.114
Val loss: 0.658 Val accuracy : 78.738
Saving best model !
******************************************
[18 epoch,   200 batch] Train loss: 0.648 Train accuracy : 83.966
Val loss: 0.781 Val accuracy : 74.647
******************************************
[19 epoch,   100 batch] Train loss: 0.628 Train accuracy : 72.963
Val loss: 0.792 Val accuracy : 73.155
******************************************
[19 epoch,   200 batch] Train loss: 0.587 Train accuracy : 78.394
Val loss: 0.647 Val accuracy : 77.408
******************************************
[20 epoch,   100 batch] Train loss: 0.591 Train accuracy : 80.033
Val loss: 0.630 Val accuracy : 79.312
Saving best model !
******************************************
[20 epoch,   200 batch] Train loss: 0.579 Train accuracy : 75.845
Val loss: 0.752 Val accuracy : 77.190
******************************************
Finish Training
```

7. Model training summary plots

In [53]:

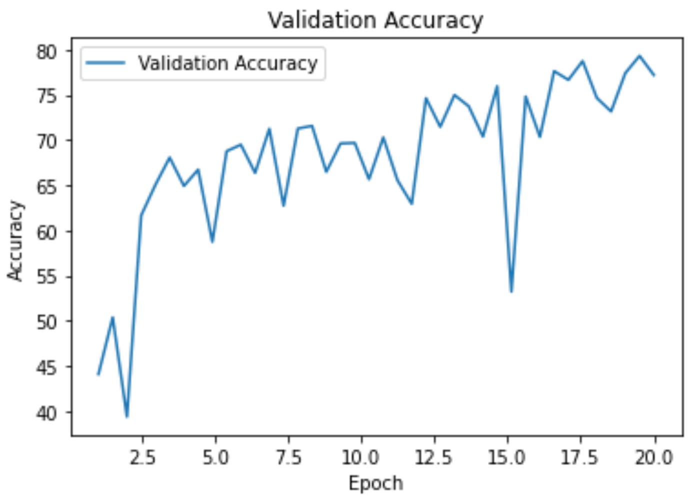
```
epoch_array = np.linspace(1, epochs, len(train_loss_array))
plt.plot(epoch_array, train_loss_array, label = "Train Loss")
plt.plot(epoch_array, val_loss_array, label = "Validation Loss")
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss Curve')
plt.legend()
# Display a figure.
plt.show()
```



From the plot we can see that the training loss and validation loss are decreasing swiftly during training of the model. The validation loss is changing within the entire process however the overall nature of the curve is decreasing. And the performance has not saturated yet, so we can still improve the performance if we train for more epoch or by changing the hyperparameters.

In [54]:

```
plt.plot(epoch_array, val_acc_array, label = "Validation Accuracy")
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Validation Accuracy')
plt.legend()
# Display a figure.
plt.show()
```



The above curve shows the validation accuracy performance during the training process.

8. Model Testing

First the best model parameter is loaded for the inferencing.

In [61]:

```
# Best validation accuracy Unet Model
PATH = 'model_best.pth.tar'
state = torch.load(PATH)

net = UNet(n_channels=3, n_classes=6, bilinear=True)
net.load_state_dict(state['state_dict'])
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Using {}".format(device))
net = net.to(device=device)
```

```
Using cuda
```

Test step

In [62]:

```
test_loss = 0.0
test_accuracy = 0.0
n_test = len(testloader)
with torch.no_grad():
    net.eval()
    for data in testloader:
        images = data[0]
        true_masks = data[1]
        images = images.to(device=device, dtype=torch.float32)
        true_masks = true_masks.to(device=device)

        masks_pred = net(images)
        loss = CrossEntropy2d(masks_pred, true_masks)
        test_loss += loss.item()

        masks_pred_cpu = masks_pred.cpu().detach().numpy()
        true_masks_cpu = true_masks.cpu().detach().numpy()
        test_accuracy += metrics(np.argmax(masks_pred_cpu, axis=1).flatten(), true_masks_cpu.fla
tten(), label_values=LABELS)
```

In [63]:

```
# 20 epoch result
print("Test Loss = {}".format(test_loss/n_test))
print("Test Accuracy = {}".format(test_accuracy/n_test))
```

```
Test Loss = 0.5495890513062477
Test Accuracy = 79.9924
```

9. Generating the prediction results and ground truth

Code snippet for generating the prediction plots

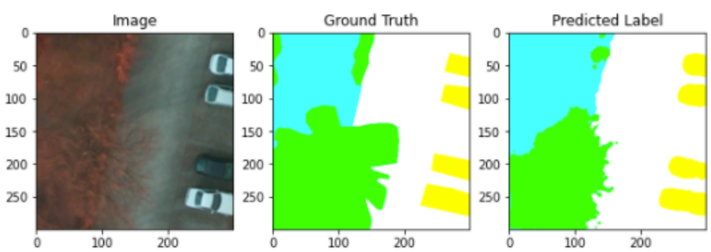
```
dataiter = iter(testloader)
```

In [79]:
```
images, labels = next(dataiter)

# UNET Model
with torch.no_grad():
    net.eval()
    net.to(device=device) # transferring to GPU

    images = images.to(device=device, dtype=torch.float32)
    labels = labels.to(device=device)
    output_pred = net(images)
```

In [80]:
```
output_pred_cpu = output_pred.cpu().detach().numpy()
labels_cpu = labels.cpu().detach().numpy()
```

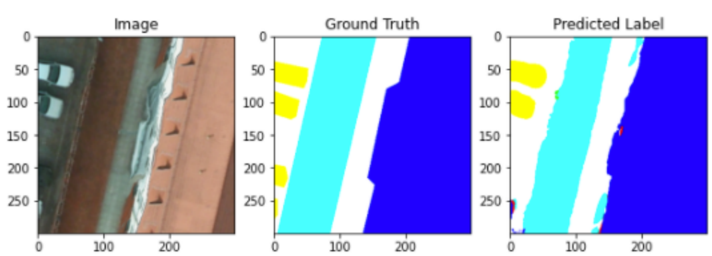
In [81]:
```
# Model prediction on data sample 1
image_number = 0
print("Prediction from model of accuracy %0.2f"%state['best_acc1'])
color_label = convert_to_color(labels[image_number].cpu().detach().numpy())
pred_color_label = convert_to_color(np.argmax(output_pred_cpu, axis=1)[image_number])
show_image_and_label(images[image_number].cpu().detach().numpy(), color_label, pred_color_label)
```

```
Prediction from model of accuracy 79.31
```




```
# Model prediction on data sample 2
image_number = 1
print("Prediction from model of accuracy %0.2f"%state['best_acc1'])
color_label = convert_to_color(labels[image_number].cpu().detach().numpy())
pred_color_label = convert_to_color(np.argmax(output_pred_cpu, axis=1)[image_number])
show_image_and_label(images[image_number].cpu().detach().numpy(), color_label, pred_color_label)
```

```
Prediction from model of accuracy 79.31
```



```
# Model prediction data sample 3
image_number = 3
print("Prediction from model of accuracy %0.2f"%state['best_acc1'])
color_label = convert_to_color(labels[image_number].cpu().detach().numpy())
pred_color_label = convert_to_color(np.argmax(output_pred_cpu, axis=1)[image_number])
show_image_and_label(images[image_number].cpu().detach().numpy(), color_label, pred_color_label)
```

```
Prediction from model of accuracy 79.31
```