Adapted from https://www.learnopencv.com/image-classification-using-transfer-learning-in-pytorch/ (https://www.learnopencv.com/image-classification-using-transfer-learning-in-pytorch/)

In [16]:
```
from __future__ import print_function, division
import os
import torch
import torch.nn as nn
import torch.optim as optim
import pandas as pd
from skimage import io, transform
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, utils, datasets, models
import time
```

In [2]:
```
# Applying Transforms to the Data
image_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(size=256, scale=(0.8, 1.0)),
        transforms.RandomRotation(degrees=15),
        transforms.RandomHorizontalFlip(),
        transforms.CenterCrop(size=224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    'valid': transforms.Compose([
        transforms.Resize(size=256),
        transforms.CenterCrop(size=224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(size=256),
        transforms.CenterCrop(size=224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ])
}
```

In [3]:
```
# Load the Data

# Set train and valid directory paths
train_directory = '../../car-images/Train'
valid_directory = '../../car-images/Validate'
test_directory = '../../car-images/Test'

# Batch size
bs = 32

# Number of classes (front of car or not)
num_classes = 2

# Load Data from folders
data = {
    'train': datasets.ImageFolder(root=train_directory, transform=image_
transforms['train']),
    'valid': datasets.ImageFolder(root=valid_directory, transform=image_
transforms['valid']),
    'test': datasets.ImageFolder(root=test_directory, transform=image_tr
ansforms['test'])
}

# Size of Data, to be used for calculating Average Loss and Accuracy
train_data_size = len(data['train'])
valid_data_size = len(data['valid'])
test_data_size = len(data['test'])

# Create iterators for the Data loaded using DataLoader module
train_data = DataLoader(data['train'], batch_size=bs, shuffle=True)
valid_data = DataLoader(data['valid'], batch_size=bs, shuffle=True)
test_data = DataLoader(data['test'], batch_size=bs, shuffle=True)

# Print the train, validation and test set data sizes
train_data_size, valid_data_size, test_data_size
```

Out[3]:
```
(1499, 250, 1036)
```

In [6]:
```
# Load pretrained ResNet50 Model
resnet50 = models.resnet50(pretrained=True)
```

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pt
h" to /home/cardetector/.cache/torch/checkpoints/resnet50-19c8e357.pth
100%|██████████| 97.8M/97.8M [00:00<00:00, 104MB/s]
```

Canziani et al. (https://arxiv.org/pdf/1605.07678.pdf (https://arxiv.org/pdf/1605.07678.pdf)) list many pretrained models that are used for various practical applications, analyzing the accuracy obtained and the inference time needed for each model. ResNet50 is one of those having a good tradeoff between accuracy and inference time. When a model is loaded in PyTorch, all its parameters have their ‘requires_grad‘ field set to true by default. That means each and every change to the parameter values will be stored in order to be used in the back propagation graph used for training. This increases memory requirements. So, since most of the parameters in our pre-trained model are already trained for us, we reset the requires_grad field to false.

In [7]:
```
# Freeze model parameters
for param in resnet50.parameters():
    param.requires_grad = False
```

Then we replace the final layer of the ResNet50 model by a small set of Sequential layers. The inputs to the last fully connected layer of ResNet50 is fed to a Linear layer which has 256 outputs, which are then fed into ReLU and Dropout layers. It is then followed by a 256×10 Linear Layer which has 2 outputs corresponding to the 2 classes in our car image data.

In [10]:
```
# Change the final layer of ResNet50 Model for Transfer Learning
fc_inputs = resnet50.fc.in_features

resnet50.fc = nn.Sequential(
    nn.Linear(fc_inputs, 256),
    nn.ReLU(),
    nn.Dropout(0.4),
    nn.Linear(256, 2),
    nn.LogSoftmax(dim=1) # For using NLLLoss()
)
```

In [12]:
```
# Define Optimizer and Loss Function
loss_func = nn.NLLLoss()
optimizer = optim.Adam(resnet50.parameters())
```

In [18]:
```
def train_and_validate(model, loss_criterion, optimizer, epochs=25):
    '''
    Function to train and validate
    Parameters
        :param model: Model to train and validate
        :param loss_criterion: Loss Criterion to minimize
        :param optimizer: Optimizer for computing gradients
        :param epochs: Number of epochs (default=25)

    Returns
        model: Trained Model with best validation accuracy
        history: (dict object): Having training loss, accuracy and valid
ation loss, accuracy
    '''

    start = time.time()
    history = []
    best_acc = 0.0

    for epoch in range(epochs):
        epoch_start = time.time()
        print("Epoch: {}/{}".format(epoch+1, epochs))

        # Set to training mode
        model.train()

        # Loss and Accuracy within the epoch
        train_loss = 0.0
        train_acc = 0.0

        valid_loss = 0.0
        valid_acc = 0.0

        for i, (inputs, labels) in enumerate(train_data):

            inputs = inputs.to(device)
            labels = labels.to(device)

            # Clean existing gradients
            optimizer.zero_grad()

            # Forward pass - compute outputs on input data using the mod
el
            outputs = model(inputs)

            # Compute loss
            loss = loss_criterion(outputs, labels)

            # Backpropagate the gradients
            loss.backward()

            # Update the parameters
            optimizer.step()

            # Compute the total loss for the batch and add it to train_l
oss
```

```
            train_loss += loss.item() * inputs.size(0)

            # Compute the accuracy
            ret, predictions = torch.max(outputs.data, 1)
            correct_counts = predictions.eq(labels.data.view_as(predicti
ons))

            # Convert correct_counts to float and then compute the mean
            acc = torch.mean(correct_counts.type(torch.FloatTensor))

            # Compute total accuracy in the whole batch and add to train
_acc
            train_acc += acc.item() * inputs.size(0)

            #print("Batch number: {:03d}, Training: Loss: {:.4f}, Accura
cy: {:.4f}".format(i, loss.item(), acc.item()))


        # Validation - No gradient tracking needed
        with torch.no_grad():

            # Set to evaluation mode
            model.eval()

            # Validation loop
            for j, (inputs, labels) in enumerate(valid_data):
                inputs = inputs.to(device)
                labels = labels.to(device)

                # Forward pass - compute outputs on input data using the
model
                outputs = model(inputs)

                # Compute loss
                loss = loss_criterion(outputs, labels)

                # Compute the total loss for the batch and add it to val
id_loss
                valid_loss += loss.item() * inputs.size(0)

                # Calculate validation accuracy
                ret, predictions = torch.max(outputs.data, 1)
                correct_counts = predictions.eq(labels.data.view_as(pred
ictions))

                # Convert correct_counts to float and then compute the m
ean
                acc = torch.mean(correct_counts.type(torch.FloatTensor))

                # Compute total accuracy in the whole batch and add to v
alid_acc
                valid_acc += acc.item() * inputs.size(0)

                #print("Validation Batch number: {:03d}, Validation: Los
s: {:.4f}, Accuracy: {:.4f}".format(j, loss.item(), acc.item()))

        # Find average training loss and training accuracy
```

```
        avg_train_loss = train_loss/train_data_size
        avg_train_acc = train_acc/train_data_size

        # Find average training loss and training accuracy
        avg_valid_loss = valid_loss/valid_data_size
        avg_valid_acc = valid_acc/valid_data_size

        history.append([avg_train_loss, avg_valid_loss, avg_train_acc, a
vg_valid_acc])

        epoch_end = time.time()

        print("Epoch : {:03d}, Training: Loss: {:.4f}, Accuracy: {:.4f}
%, \n\t\tValidation : Loss : {:.4f}, Accuracy: {:.4f}%, Time: {:.4f}s".f
ormat(epoch, avg_train_loss, avg_train_acc*100, avg_valid_loss, avg_vali
d_acc*100, epoch_end-epoch_start))

        # Save if the model has best accuracy till now
        torch.save(model, dataset+'_model_'+str(epoch)+'.pt')

    return model, history
```

In [20]:
```
dataset = 'car-images'

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Print the model to be trained
#summary(resnet50, input_size=(3, 224, 224), batch_size=bs, device='cud
a')

# Train the model for 25 epochs
num_epochs = 30
trained_model, history = train_and_validate(resnet50, loss_func, optimiz
er, num_epochs)

torch.save(history, dataset+'_history.pt')
```

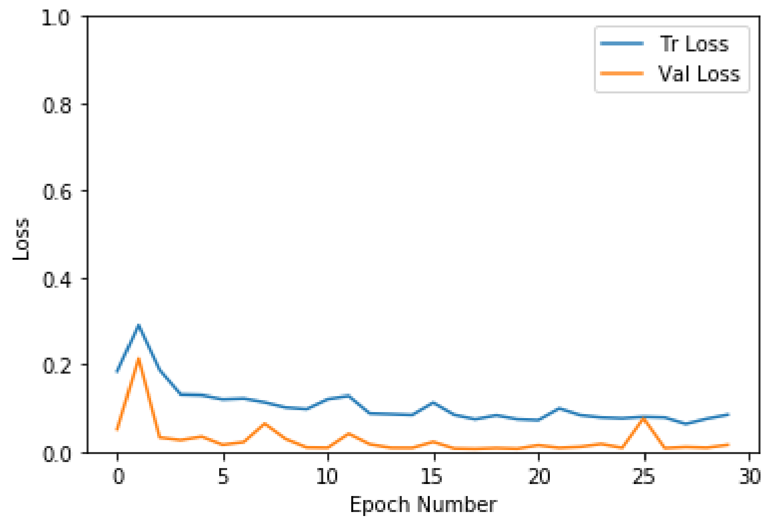
```
Epoch: 1/30
Epoch : 000, Training: Loss: 0.1841, Accuracy: 92.3282%, 
		Validation : Loss : 0.0517, Accuracy: 100.0000%, Time: 
150.4628s
Epoch: 2/30
Epoch : 001, Training: Loss: 0.2901, Accuracy: 87.4583%, 
		Validation : Loss : 0.2134, Accuracy: 89.2000%, Time: 1
50.2779s
Epoch: 3/30
Epoch : 002, Training: Loss: 0.1869, Accuracy: 92.4616%, 
		Validation : Loss : 0.0327, Accuracy: 100.0000%, Time: 
147.7697s
Epoch: 4/30
Epoch : 003, Training: Loss: 0.1309, Accuracy: 94.7965%, 
		Validation : Loss : 0.0266, Accuracy: 100.0000%, Time: 
153.2258s
Epoch: 5/30
Epoch : 004, Training: Loss: 0.1296, Accuracy: 94.9967%, 
		Validation : Loss : 0.0347, Accuracy: 100.0000%, Time: 
146.6597s
Epoch: 6/30
Epoch : 005, Training: Loss: 0.1202, Accuracy: 95.5304%, 
		Validation : Loss : 0.0158, Accuracy: 100.0000%, Time: 
147.1634s
Epoch: 7/30
Epoch : 006, Training: Loss: 0.1222, Accuracy: 95.1968%, 
		Validation : Loss : 0.0221, Accuracy: 100.0000%, Time: 
148.2505s
Epoch: 8/30
Epoch : 007, Training: Loss: 0.1135, Accuracy: 94.8632%, 
		Validation : Loss : 0.0645, Accuracy: 96.8000%, Time: 1
47.2192s
Epoch: 9/30
Epoch : 008, Training: Loss: 0.1016, Accuracy: 96.3309%, 
		Validation : Loss : 0.0293, Accuracy: 100.0000%, Time: 
147.5290s
Epoch: 10/30
Epoch : 009, Training: Loss: 0.0979, Accuracy: 96.1975%, 
		Validation : Loss : 0.0097, Accuracy: 100.0000%, Time: 
147.1863s
Epoch: 11/30
Epoch : 010, Training: Loss: 0.1207, Accuracy: 94.9967%, 
		Validation : Loss : 0.0090, Accuracy: 100.0000%, Time: 
147.9490s
Epoch: 12/30
Epoch : 011, Training: Loss: 0.1280, Accuracy: 94.3963%, 
		Validation : Loss : 0.0415, Accuracy: 98.0000%, Time: 1
46.9841s
Epoch: 13/30
Epoch : 012, Training: Loss: 0.0880, Accuracy: 96.9313%, 
		Validation : Loss : 0.0172, Accuracy: 100.0000%, Time: 
147.1734s
Epoch: 14/30
Epoch : 013, Training: Loss: 0.0863, Accuracy: 96.6644%, 
		Validation : Loss : 0.0088, Accuracy: 100.0000%, Time: 
147.8408s
Epoch: 15/30
```

```
Epoch : 014, Training: Loss: 0.0844, Accuracy: 96.7979%, 
		Validation : Loss : 0.0087, Accuracy: 100.0000%, Time: 
147.8941s
Epoch: 16/30
Epoch : 015, Training: Loss: 0.1129, Accuracy: 95.3969%, 
		Validation : Loss : 0.0227, Accuracy: 99.6000%, Time: 1
47.3496s
Epoch: 17/30
Epoch : 016, Training: Loss: 0.0848, Accuracy: 97.1981%, 
		Validation : Loss : 0.0074, Accuracy: 100.0000%, Time: 
147.1528s
Epoch: 18/30
Epoch : 017, Training: Loss: 0.0746, Accuracy: 97.1314%, 
		Validation : Loss : 0.0065, Accuracy: 100.0000%, Time: 
147.6004s
Epoch: 19/30
Epoch : 018, Training: Loss: 0.0837, Accuracy: 96.9313%, 
		Validation : Loss : 0.0083, Accuracy: 100.0000%, Time: 
147.7589s
Epoch: 20/30
Epoch : 019, Training: Loss: 0.0748, Accuracy: 97.1314%, 
		Validation : Loss : 0.0068, Accuracy: 100.0000%, Time: 
148.7488s
Epoch: 21/30
Epoch : 020, Training: Loss: 0.0726, Accuracy: 97.4650%, 
		Validation : Loss : 0.0148, Accuracy: 100.0000%, Time: 
148.5910s
Epoch: 22/30
Epoch : 021, Training: Loss: 0.0996, Accuracy: 96.3976%, 
		Validation : Loss : 0.0088, Accuracy: 100.0000%, Time: 
148.9475s
Epoch: 23/30
Epoch : 022, Training: Loss: 0.0840, Accuracy: 96.7312%, 
		Validation : Loss : 0.0112, Accuracy: 100.0000%, Time: 
148.6319s
Epoch: 24/30
Epoch : 023, Training: Loss: 0.0787, Accuracy: 97.3316%, 
		Validation : Loss : 0.0178, Accuracy: 99.6000%, Time: 1
48.9003s
Epoch: 25/30
Epoch : 024, Training: Loss: 0.0768, Accuracy: 97.1981%, 
		Validation : Loss : 0.0088, Accuracy: 100.0000%, Time: 
149.5885s
Epoch: 26/30
Epoch : 025, Training: Loss: 0.0803, Accuracy: 96.4643%, 
		Validation : Loss : 0.0763, Accuracy: 98.0000%, Time: 1
48.7704s
Epoch: 27/30
Epoch : 026, Training: Loss: 0.0788, Accuracy: 96.7312%, 
		Validation : Loss : 0.0082, Accuracy: 100.0000%, Time: 
148.0588s
Epoch: 28/30
Epoch : 027, Training: Loss: 0.0638, Accuracy: 97.7318%, 
		Validation : Loss : 0.0108, Accuracy: 100.0000%, Time: 
149.0482s
Epoch: 29/30
Epoch : 028, Training: Loss: 0.0758, Accuracy: 96.9980%, 
```

```
                   Validation : Loss : 0.0091, Accuracy: 100.0000%, Time:
153.0935s
Epoch: 30/30
Epoch : 029, Training: Loss: 0.0851, Accuracy: 96.9313%,
                   Validation : Loss : 0.0156, Accuracy: 99.2000%, Time: 1
48.4307s
```

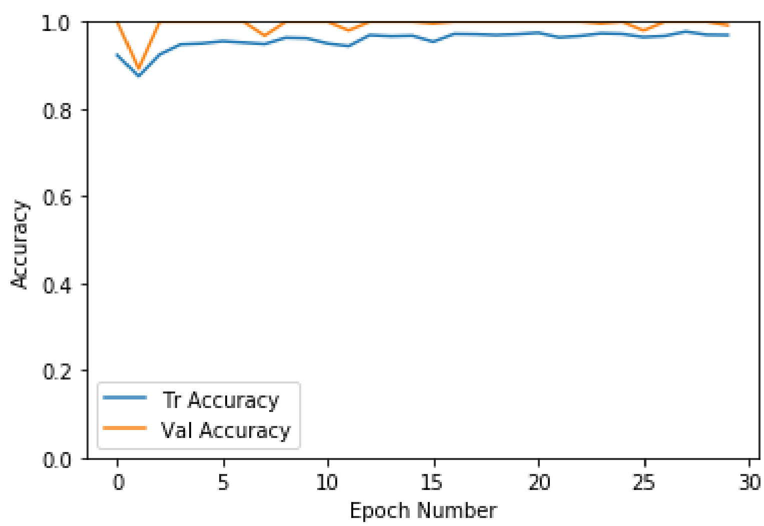
In [21]:

```
history = np.array(history)
plt.plot(history[:,0:2])
plt.legend(['Tr Loss', 'Val Loss'])
plt.xlabel('Epoch Number')
plt.ylabel('Loss')
plt.ylim(0,1)
plt.savefig(dataset+'_loss_curve.png')
plt.show()
```



In [22]:

```
plt.plot(history[:,2:4])
plt.legend(['Tr Accuracy', 'Val Accuracy'])
plt.xlabel('Epoch Number')
plt.ylabel('Accuracy')
plt.ylim(0,1)
plt.savefig(dataset+'_accuracy_curve.png')
plt.show()
```



In [25]:
```
def computeTestSetAccuracy(model, loss_criterion):
    '''
    Function to compute the accuracy on the test set
    Parameters
        :param model: Model to test
        :param loss_criterion: Loss Criterion to minimize
    '''

    device = torch.device("cuda:0" if torch.cuda.is_available() else "cp
u")

    test_acc = 0.0
    test_loss = 0.0

    # Validation - No gradient tracking needed
    with torch.no_grad():

        # Set to evaluation mode
        model.eval()

        # Validation loop
        for j, (inputs, labels) in enumerate(test_data):
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Forward pass - compute outputs on input data using the mod
el
            outputs = model(inputs)

            # Compute loss
            loss = loss_criterion(outputs, labels)

            # Compute the total loss for the batch and add it to valid_l
oss
            test_loss += loss.item() * inputs.size(0)

            # Calculate validation accuracy
            ret, predictions = torch.max(outputs.data, 1)
            correct_counts = predictions.eq(labels.data.view_as(predicti
ons))

            # Convert correct_counts to float and then compute the mean
            acc = torch.mean(correct_counts.type(torch.FloatTensor))

            # Compute total accuracy in the whole batch and add to valid
_acc
            test_acc += acc.item() * inputs.size(0)

            print("Test Batch number: {:03d}, Test: Loss: {:.4f}, Accura
cy: {:.4f}".format(j, loss.item(), acc.item()))

    # Find average test loss and test accuracy
    avg_test_loss = test_loss/test_data_size
    avg_test_acc = test_acc/test_data_size

    print("Test accuracy : " + str(avg_test_acc))
```

In [26]:
```
computeTestSetAccuracy(trained_model, loss_func)
```

```
Test Batch number: 000, Test: Loss: 0.4020, Accuracy: 0.8750
Test Batch number: 001, Test: Loss: 0.1053, Accuracy: 0.9375
Test Batch number: 002, Test: Loss: 0.0120, Accuracy: 1.0000
Test Batch number: 003, Test: Loss: 0.2840, Accuracy: 0.9375
Test Batch number: 004, Test: Loss: 0.2106, Accuracy: 0.9062
Test Batch number: 005, Test: Loss: 0.4999, Accuracy: 0.7812
Test Batch number: 006, Test: Loss: 0.4529, Accuracy: 0.8438
Test Batch number: 007, Test: Loss: 0.4466, Accuracy: 0.8750
Test Batch number: 008, Test: Loss: 0.6579, Accuracy: 0.7500
Test Batch number: 009, Test: Loss: 0.2957, Accuracy: 0.9062
Test Batch number: 010, Test: Loss: 0.6570, Accuracy: 0.7812
Test Batch number: 011, Test: Loss: 0.3110, Accuracy: 0.8438
Test Batch number: 012, Test: Loss: 0.5766, Accuracy: 0.8438
Test Batch number: 013, Test: Loss: 0.3161, Accuracy: 0.8750
Test Batch number: 014, Test: Loss: 0.2694, Accuracy: 0.8750
Test Batch number: 015, Test: Loss: 0.3147, Accuracy: 0.8438
Test Batch number: 016, Test: Loss: 0.0972, Accuracy: 0.9375
Test Batch number: 017, Test: Loss: 0.8037, Accuracy: 0.7812
Test Batch number: 018, Test: Loss: 0.3123, Accuracy: 0.8750
Test Batch number: 019, Test: Loss: 0.4241, Accuracy: 0.8438
Test Batch number: 020, Test: Loss: 0.2741, Accuracy: 0.9062
Test Batch number: 021, Test: Loss: 0.3304, Accuracy: 0.9062
Test Batch number: 022, Test: Loss: 0.6098, Accuracy: 0.8125
Test Batch number: 023, Test: Loss: 0.5335, Accuracy: 0.7812
Test Batch number: 024, Test: Loss: 0.5496, Accuracy: 0.8438
Test Batch number: 025, Test: Loss: 0.2427, Accuracy: 0.8438
Test Batch number: 026, Test: Loss: 0.2871, Accuracy: 0.9062
Test Batch number: 027, Test: Loss: 0.5885, Accuracy: 0.8125
Test Batch number: 028, Test: Loss: 0.3036, Accuracy: 0.9062
Test Batch number: 029, Test: Loss: 0.5065, Accuracy: 0.8750
Test Batch number: 030, Test: Loss: 0.6527, Accuracy: 0.9062
Test Batch number: 031, Test: Loss: 0.1970, Accuracy: 0.9375
Test Batch number: 032, Test: Loss: 0.8161, Accuracy: 0.7500
Test accuracy : 0.8658301158301158
```

In [45]:
```
from PIL import Image

def predict(model, test_image_name):
    '''
    Function to predict the class of a single test image
    Parameters
        :param model: Model to test
        :param test_image_name: Test image

    '''

    transform = image_transforms['test']

    test_image = Image.open(test_image_name)
    plt.imshow(test_image)

    test_image_tensor = transform(test_image)

    if torch.cuda.is_available():
        test_image_tensor = test_image_tensor.view(1, 3, 224, 224).cuda
()
    else:
        test_image_tensor = test_image_tensor.view(1, 3, 224, 224)

    with torch.no_grad():
        model.eval()
        # Model outputs log probabilities
        out = model(test_image_tensor)
        ps = torch.exp(out)
        topk, topclass = ps.topk(2, dim=1)

        # Convert indices to classes
        idx_to_class = {val: key for key, val in
                                       data['test'].class_to_idx.items()}

        for i in range(2):
            print("Prediction", i+1, ":", idx_to_class[topclass.cpu().nu
mpy()[0][i]], ", Score: ", topk.cpu().numpy()[0][i])
```

In [46]:
```
# Test a particular model on a test image


model = torch.load(dataset+'_model_29.pt')
# model_number.pt, where number = epoch
predict(model, '../../car-images/Test/Front-View/tata16.jpg')

# Load Data from folders
#computeTestSetAccuracy(model, loss_func)
```

```
Prediction 1 : Front-View , Score:  0.9999746
Prediction 2 : Not-Front-View , Score:  2.540321e-05
```

In [47]:
```
predict(model, '../../car-images/Test/Not-Front-View/tripod_seq_02_094.j
pg')
```

```
Prediction 1 : Not-Front-View , Score:  0.9999099
Prediction 2 : Front-View , Score:  9.0080925e-05
```