Transfer Learning

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import numpy as np
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy
import pretrainedmodels

torch.manual_seed(0)      # for reproducibility
torch.cuda.empty_cache()
```

Load Data

We will use torchvision and torch.utils.data packages for loading the data.

In [2]:
```
# Data augmentation and normalization for training
# Just normalization for validation
data_transforms = {
    'train': transforms.Compose([
        transforms.Resize([224, 224]),
        transforms.ToTensor()
    ]),
    'val': transforms.Compose([
        transforms.Resize([224, 224]),
        transforms.ToTensor()
    ]),
    'test': transforms.Compose([
        transforms.Resize([224, 224]),
        transforms.ToTensor()
    ])
}

data_dir = 'D:\\data (unaugmented, 2 classes, tif)'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'val', 'test']}

batch_size = 128       # Need it as a global variable for computing average loss/accuracy per iterati
on
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size,
                                             shuffle=True, num_workers=4)
              for x in ['train', 'val', 'test']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val', 'test']}
class_names = image_datasets['train'].classes

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

Training the model

Now, let's write a general function to train a model. Here, we will illustrate:

- Scheduling the learning rate
- Saving the best model

In the following, parameter `scheduler` is an LR scheduler object from `torch.optim.lr_scheduler`.

In [3]:
```
def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    epoch_numbers = []
    epoch_train_accuracies = []
    epoch_train_losses = []
    epoch_val_accuracies = []
    epoch_val_losses = []

    for epoch in range(num_epochs):
        epoch_numbers.append(epoch)              # for plotting
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                if scheduler != "":
                    scheduler.step()
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:      # The labels will correspond to the alpha
betical order of the class names (https://discuss.pytorch.org/t/how-to-get-the-class-names-to-class-
label-mapping/470).
                inputs = inputs.to(device)
                labels = labels.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss/dataset_sizes[phase]
            epoch_acc = running_corrects.double()/dataset_sizes[phase]

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))

            # For plotting
            if phase == 'train':
                epoch_train_accuracies.append(epoch_acc)
                epoch_train_losses.append(epoch_loss)
            else:
                epoch_val_accuracies.append(epoch_acc)
                epoch_val_losses.append(epoch_loss)

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

        print()

    # Plotting

    plt.title("Training Curve (Loss)")
    plt.plot(epoch_numbers, epoch_train_losses, label="Train")
    plt.plot(epoch_numbers, epoch_val_losses, label="Validation")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend(loc='best')
    plt.show()

    plt.title("Training Curve (Accuracy)")
    plt.plot(epoch_numbers, epoch_train_accuracies, label="Train")
    plt.plot(epoch_numbers, epoch_val_accuracies, label="Validation")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

    # load best model weights
    model.load_state_dict(best_model_wts)
    return model
```

Finetuning the convnet

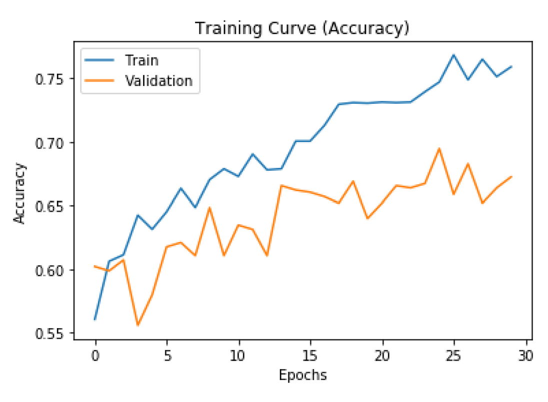
Load a pretrained model and reset final fully connected layer.

In [4]:
```
model_ft = models.alexnet(pretrained=True)
model_ft.classifier[6] = nn.Linear(4096, len(class_names))
model_ft.classifier.add_module("7", nn.Dropout())

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
#optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)

optimizer_ft = optim.Adam([
                {'params': model_ft.features.parameters(), 'lr': 0.0001},        # The other (non-fi
nal) layers will have a lr = 0.1*base lr.
                {'params': model_ft.classifier[:6].parameters(), 'lr': 0.0001},
                {'params': model_ft.classifier[6:].parameters()}                 # The final layers
 will have the base lr.
            ], lr=0.001, weight_decay=0.005)
```

Train and evaluate

In [5]:
```
model_ft = train_model(model_ft, criterion, optimizer_ft, scheduler="",
                       num_epochs=30)
```

```
Epoch 0/29
----------
train Loss: 0.7239 Acc: 0.5604
val Loss: 0.6472 Acc: 0.6021

Epoch 1/29
----------
train Loss: 0.6450 Acc: 0.6062
val Loss: 0.6351 Acc: 0.5986

Epoch 2/29
----------
train Loss: 0.6271 Acc: 0.6113
val Loss: 0.6340 Acc: 0.6072

Epoch 3/29
----------
train Loss: 0.6228 Acc: 0.6425
val Loss: 0.6351 Acc: 0.5557

Epoch 4/29
----------
train Loss: 0.6256 Acc: 0.6314
val Loss: 0.6281 Acc: 0.5798

Epoch 5/29
----------
train Loss: 0.6128 Acc: 0.6450
val Loss: 0.6326 Acc: 0.6175

Epoch 6/29
----------
train Loss: 0.6018 Acc: 0.6634
val Loss: 0.6531 Acc: 0.6209

Epoch 7/29
----------
train Loss: 0.5966 Acc: 0.6484
val Loss: 0.6706 Acc: 0.6106

Epoch 8/29
----------
train Loss: 0.5870 Acc: 0.6702
val Loss: 0.6557 Acc: 0.6484

Epoch 9/29
----------
train Loss: 0.5842 Acc: 0.6788
val Loss: 0.6296 Acc: 0.6106

Epoch 10/29
----------
train Loss: 0.5781 Acc: 0.6728
val Loss: 0.6412 Acc: 0.6346

Epoch 11/29
----------
train Loss: 0.5695 Acc: 0.6903
val Loss: 0.6377 Acc: 0.6312

Epoch 12/29
----------
train Loss: 0.5746 Acc: 0.6779
val Loss: 0.6179 Acc: 0.6106

Epoch 13/29
----------
train Loss: 0.5671 Acc: 0.6788
val Loss: 0.6178 Acc: 0.6655

Epoch 14/29
----------
train Loss: 0.5679 Acc: 0.7006
val Loss: 0.6257 Acc: 0.6621

Epoch 15/29
----------
train Loss: 0.5465 Acc: 0.7006
val Loss: 0.6702 Acc: 0.6604

Epoch 16/29
----------
train Loss: 0.5417 Acc: 0.7129
val Loss: 0.6619 Acc: 0.6569

Epoch 17/29
----------
train Loss: 0.5229 Acc: 0.7296
val Loss: 0.7419 Acc: 0.6518

Epoch 18/29
----------
train Loss: 0.5289 Acc: 0.7309
val Loss: 0.6806 Acc: 0.6690

Epoch 19/29
----------
train Loss: 0.5242 Acc: 0.7305
val Loss: 0.6997 Acc: 0.6398

Epoch 20/29
----------
train Loss: 0.5306 Acc: 0.7313
val Loss: 0.6393 Acc: 0.6518

Epoch 21/29
----------
train Loss: 0.5115 Acc: 0.7309
val Loss: 0.7025 Acc: 0.6655

Epoch 22/29
----------
train Loss: 0.5117 Acc: 0.7313
val Loss: 0.6184 Acc: 0.6638

Epoch 23/29
----------
train Loss: 0.5119 Acc: 0.7394
val Loss: 0.6449 Acc: 0.6672

Epoch 24/29
----------
train Loss: 0.4834 Acc: 0.7471
val Loss: 0.6470 Acc: 0.6947

Epoch 25/29
----------
train Loss: 0.4754 Acc: 0.7685
val Loss: 0.7224 Acc: 0.6587

Epoch 26/29
----------
train Loss: 0.4888 Acc: 0.7488
val Loss: 0.7038 Acc: 0.6827

Epoch 27/29
----------
train Loss: 0.4826 Acc: 0.7651
val Loss: 0.7541 Acc: 0.6518

Epoch 28/29
----------
train Loss: 0.4731 Acc: 0.7514
val Loss: 0.6731 Acc: 0.6638

Epoch 29/29
----------
train Loss: 0.4805 Acc: 0.7591
val Loss: 0.6890 Acc: 0.6724
```




```
Training complete in 4m 47s
Best val Acc: 0.694683
```

In [6]:
```
dt = time.strftime("%Y%m%d-%H%M%S")
dt
```

Out[6]: '20190409-180435'

In [7]:
```
# Save best model to disk for later!
torch.save(model_ft.state_dict(), os.getcwd() + '\\' + 'model_' + dt + '.pth')
```

In [8]:
```
# Play sound when code finishes.

import winsound
duration = 1500  # milliseconds
```