Transfer Learning

In [9]:
```python
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 [10]:
```python
# 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 [11]:
```python
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

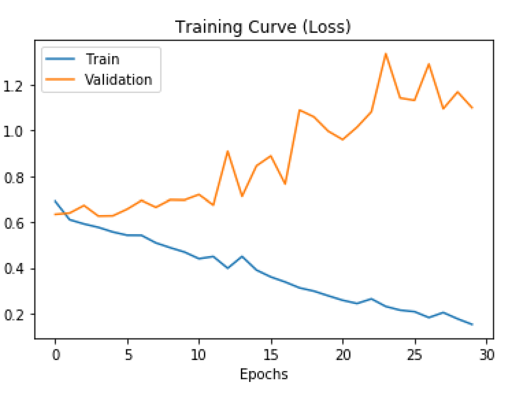
Load a pretrained model and reset final fully connected layer.

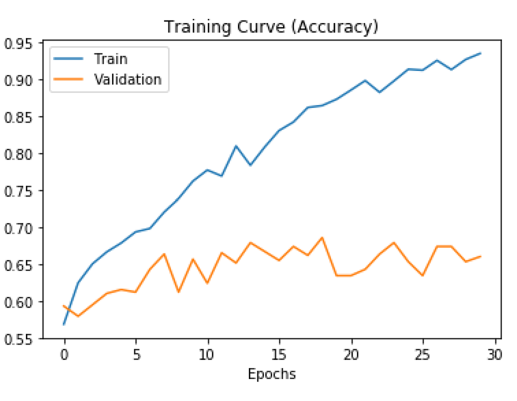
In [12]:
```python
model_ft = models.alexnet(pretrained=True)
model_ft.classifier[6] = nn.Linear(4096, len(class_names))
#model_ft.classifier.add_module("7", nn.Dropout())

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 [13]:
```python
model_ft = train_model(model_ft, criterion, optimizer_ft, scheduler="",
                       num_epochs=30)
```

```
Epoch 0/29
----------
train Loss: 0.6907 Acc: 0.5690
val Loss: 0.6342 Acc: 0.5935

Epoch 1/29
----------
train Loss: 0.6109 Acc: 0.6249
val Loss: 0.6398 Acc: 0.5798

Epoch 2/29
----------
train Loss: 0.5922 Acc: 0.6506
val Loss: 0.6729 Acc: 0.5952

Epoch 3/29
----------
train Loss: 0.5774 Acc: 0.6668
val Loss: 0.6266 Acc: 0.6106

Epoch 4/29
----------
train Loss: 0.5573 Acc: 0.6788
val Loss: 0.6276 Acc: 0.6158

Epoch 5/29
----------
train Loss: 0.5429 Acc: 0.6937
val Loss: 0.6567 Acc: 0.6123

Epoch 6/29
----------
train Loss: 0.5424 Acc: 0.6984
val Loss: 0.6946 Acc: 0.6432

Epoch 7/29
----------
train Loss: 0.5095 Acc: 0.7206
val Loss: 0.6649 Acc: 0.6638

Epoch 8/29
----------
train Loss: 0.4885 Acc: 0.7386
val Loss: 0.6974 Acc: 0.6123

Epoch 9/29
----------
train Loss: 0.4692 Acc: 0.7621
val Loss: 0.6964 Acc: 0.6569

Epoch 10/29
----------
train Loss: 0.4403 Acc: 0.7770
val Loss: 0.7197 Acc: 0.6244

Epoch 11/29
----------
train Loss: 0.4498 Acc: 0.7689
val Loss: 0.6744 Acc: 0.6655

Epoch 12/29
----------
train Loss: 0.3985 Acc: 0.8095
val Loss: 0.9090 Acc: 0.6518

Epoch 13/29
----------
train Loss: 0.4502 Acc: 0.7834
val Loss: 0.7118 Acc: 0.6792

Epoch 14/29
----------
train Loss: 0.3906 Acc: 0.8082
val Loss: 0.8449 Acc: 0.6672

Epoch 15/29
----------
train Loss: 0.3606 Acc: 0.8304
val Loss: 0.8884 Acc: 0.6552

Epoch 16/29
----------
train Loss: 0.3384 Acc: 0.8419
val Loss: 0.7655 Acc: 0.6741

Epoch 17/29
----------
train Loss: 0.3127 Acc: 0.8616
val Loss: 1.0890 Acc: 0.6621

Epoch 18/29
----------
train Loss: 0.2985 Acc: 0.8642
val Loss: 1.0593 Acc: 0.6861

Epoch 19/29
----------
train Loss: 0.2780 Acc: 0.8727
val Loss: 0.9965 Acc: 0.6346

Epoch 20/29
----------
train Loss: 0.2585 Acc: 0.8851
val Loss: 0.9599 Acc: 0.6346

Epoch 21/29
----------
train Loss: 0.2443 Acc: 0.8979
val Loss: 1.0141 Acc: 0.6432

Epoch 22/29
----------
train Loss: 0.2646 Acc: 0.8821
val Loss: 1.0816 Acc: 0.6638

Epoch 23/29
----------
train Loss: 0.2314 Acc: 0.8975
val Loss: 1.3358 Acc: 0.6792

Epoch 24/29
----------
train Loss: 0.2152 Acc: 0.9133
val Loss: 1.1418 Acc: 0.6535

Epoch 25/29
----------
train Loss: 0.2086 Acc: 0.9120
val Loss: 1.1316 Acc: 0.6346

Epoch 26/29
----------
train Loss: 0.1828 Acc: 0.9252
val Loss: 1.2902 Acc: 0.6741

Epoch 27/29
----------
train Loss: 0.2048 Acc: 0.9129
val Loss: 1.0947 Acc: 0.6741

Epoch 28/29
----------
train Loss: 0.1772 Acc: 0.9265
val Loss: 1.1683 Acc: 0.6535

Epoch 29/29
----------
train Loss: 0.1530 Acc: 0.9346
val Loss: 1.0995 Acc: 0.6604
```






```
Training complete in 4m 43s
Best val Acc: 0.686106
```

In [14]:
```python
dt = time.strftime("%Y%m%d-%H%M%S")
dt
```

Out[14]:
```
'20190409-183027'
```

In [15]:
```python
# Save best model to disk for later!
torch.save(model_ft.state_dict(), os.getcwd() + '\\' + 'model_' + dt + '.pth')
```

In [16]:
```python
# Play sound when code finishes.

import winsound
duration = 1500   # milliseconds
```