Transfer Learning

In [1]:

```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

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

Out[1]:

```
<torch._C.Generator at 0x1c2368f1810>
```

Load Data

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

In [2]:

```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()
    ])
}

data_dir = 'D:\\data (augmented, 4 classes, tif)'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'val']}

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']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
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]:

```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 [4]:

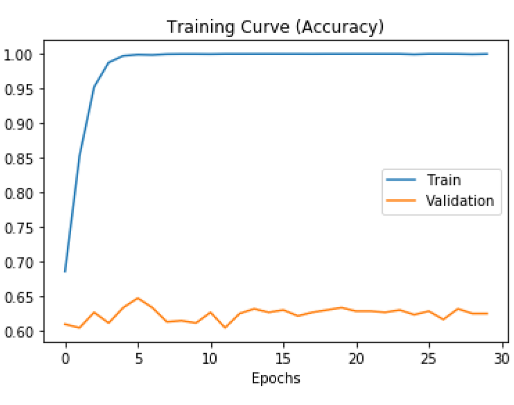
```python
model_ft = models.resnet18(pretrained=True)
model_ft.fc = nn.Linear(512, len(class_names))

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)
```

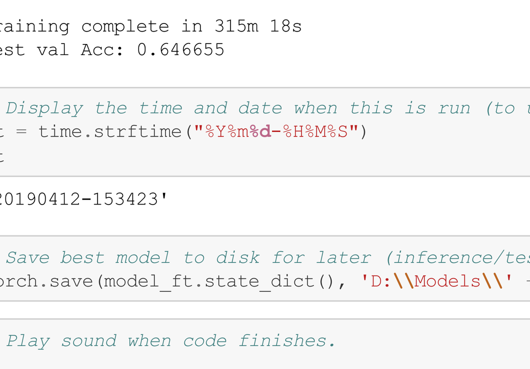
Train and evaluate

In [5]:

```python
model_ft = train_model(model_ft, criterion, optimizer_ft, scheduler="",
                       num_epochs=30)
```

```
Epoch 0/29
----------
train Loss: 0.7343 Acc: 0.6853
val Loss: 0.9156 Acc: 0.6089

Epoch 1/29
----------
train Loss: 0.3898 Acc: 0.8527
val Loss: 1.1389 Acc: 0.6038

Epoch 2/29
----------
train Loss: 0.1627 Acc: 0.9522
val Loss: 1.2487 Acc: 0.6261

Epoch 3/29
----------
train Loss: 0.0580 Acc: 0.9876
val Loss: 1.4601 Acc: 0.6106

Epoch 4/29
----------
train Loss: 0.0219 Acc: 0.9972
val Loss: 1.6251 Acc: 0.6329

Epoch 5/29
----------
train Loss: 0.0117 Acc: 0.9990
val Loss: 1.7407 Acc: 0.6467

Epoch 6/29
----------
train Loss: 0.0114 Acc: 0.9985
val Loss: 1.7856 Acc: 0.6329

Epoch 7/29
----------
train Loss: 0.0054 Acc: 0.9996
val Loss: 1.9646 Acc: 0.6123

Epoch 8/29
----------
train Loss: 0.0039 Acc: 0.9999
val Loss: 2.0178 Acc: 0.6141

Epoch 9/29
----------
train Loss: 0.0038 Acc: 0.9998
val Loss: 2.0993 Acc: 0.6106

Epoch 10/29
----------
train Loss: 0.0038 Acc: 0.9997
val Loss: 1.9956 Acc: 0.6261

Epoch 11/29
----------
train Loss: 0.0019 Acc: 1.0000
val Loss: 2.0587 Acc: 0.6038

Epoch 12/29
----------
train Loss: 0.0019 Acc: 0.9999
val Loss: 2.1454 Acc: 0.6244

Epoch 13/29
----------
train Loss: 0.0015 Acc: 1.0000
val Loss: 2.1037 Acc: 0.6312

Epoch 14/29
----------
train Loss: 0.0013 Acc: 1.0000
val Loss: 2.2034 Acc: 0.6261

Epoch 15/29
----------
train Loss: 0.0012 Acc: 0.9999
val Loss: 2.1539 Acc: 0.6295

Epoch 16/29
----------
train Loss: 0.0011 Acc: 0.9999
val Loss: 2.2132 Acc: 0.6209

Epoch 17/29
----------
train Loss: 0.0014 Acc: 0.9999
val Loss: 2.1693 Acc: 0.6261

Epoch 18/29
----------
train Loss: 0.0011 Acc: 1.0000
val Loss: 2.2416 Acc: 0.6295

Epoch 19/29
----------
train Loss: 0.0008 Acc: 1.0000
val Loss: 2.2333 Acc: 0.6329

Epoch 20/29
----------
train Loss: 0.0009 Acc: 1.0000
val Loss: 2.2446 Acc: 0.6278

Epoch 21/29
----------
train Loss: 0.0007 Acc: 1.0000
val Loss: 2.2499 Acc: 0.6278

Epoch 22/29
----------
train Loss: 0.0008 Acc: 1.0000
val Loss: 2.3004 Acc: 0.6261

Epoch 23/29
----------
train Loss: 0.0007 Acc: 0.9999
val Loss: 2.4082 Acc: 0.6295

Epoch 24/29
----------
train Loss: 0.0030 Acc: 0.9993
val Loss: 2.3500 Acc: 0.6226

Epoch 25/29
----------
train Loss: 0.0008 Acc: 1.0000
val Loss: 2.3274 Acc: 0.6278

Epoch 26/29
----------
train Loss: 0.0007 Acc: 1.0000
val Loss: 2.3406 Acc: 0.6158

Epoch 27/29
----------
train Loss: 0.0012 Acc: 0.9999
val Loss: 2.2827 Acc: 0.6312

Epoch 28/29
----------
train Loss: 0.0036 Acc: 0.9994
val Loss: 2.3203 Acc: 0.6244

Epoch 29/29
----------
train Loss: 0.0012 Acc: 0.9999
val Loss: 2.3786 Acc: 0.6244
```






```
Training complete in 315m 18s
Best val Acc: 0.646655
```

In [6]:

```python
# Display the time and date when this is run (to use for saving the model).
dt = time.strftime("%Y%m%d-%H%M%S")
dt
```

Out[6]:

```
'20190412-153423'
```

In [7]:

```python
# Save best model to disk for later (inference/testing)!
torch.save(model_ft.state_dict(), 'D:\\Models\\' + 'model_' + dt + '.pth')
```

In [8]:

```python
# Play sound when code finishes.

import winsound
duration = 1500  # milliseconds
```