Lab 3: Gesture Recognition using Convolutional Neural Networks

In this lab you will train a convolutional neural network to make classifications on different hand gestures. By the end of the lab, you should be able to:

1. Load and split data for training, validation and testing
2. Train a Convolutional Neural Network
3. Apply transfer learning to improve your model

Note that for this lab we will not be providing you with any starter code. You should be able to take the code used in previous labs, tutorials and lectures and modify it accordingly to complete the tasks outlined below.

What to submit

Submit a PDF file containing all your code, outputs, and write-up from parts 1-5. You can produce a PDF of your Google Colab file by going to **File > Print** and then save as PDF. The Colab instructions has more information. Make sure to review the PDF submission to ensure that your answers are easy to read. Make sure that your text is not cut off at the margins.

Do not submit any other files produced by your code.

Include a link to your colab file in your submission.

Please use Google Colab to complete this assignment. If you want to use Jupyter Notebook, please complete the assignment and upload your Jupyter Notebook file to Google Colab for submission.

Dataset

American Sign Language (ASL) is a complete, complex language that employs signs made by moving the hands combined with facial expressions and postures of the body. It is the primary language of many North Americans who are deaf and is one of several communication options used by people who are deaf or hard-of-hearing. The hand gestures representing English alphabet are shown below. This lab focuses on classifying a subset of these hand gesture images using convolutional neural networks. Specifically, given an image of a hand showing one of the letters A-I, we want to detect which letter is being represented.

Part B. Building a CNN [50 pt]

For this lab, we are not going to give you any starter code. You will be writing a convolutional neural network from scratch. You are welcome to use any code from previous labs, lectures and tutorials. You should also write your own code.

You may use the PyTorch documentation freely. You might also find online tutorials helpful. However, all code that you submit must be your own.

Make sure that your code is vectorized, and does not contain obvious inefficiencies (for example, unecessary for loops, or unnecessary calls to unsqueeze()). Ensure enough comments are included in the code so that your TA can understand what you are doing. It is your responsibility to show that you understand what you write.

This is much more challenging and time-consuming than the previous labs. Make sure that you give yourself plenty of time by starting early.

1. Data Loading and Splitting [5 pt]

Download the anonymized data provided on Quercus. To allow you to get a heads start on this project we will provide you with sample data from previous years. Split the data into training, validation, and test sets.

Note: Data splitting is not as trivial in this lab. We want our test set to closely resemble the setting in which our model will be used. In particular, our test set should contain hands that are never seen in training!

Explain how you split the data, either by describing what you did, or by showing the code that you used. Justify your choice of splitting strategy. How many training, validation, and test images do you have?

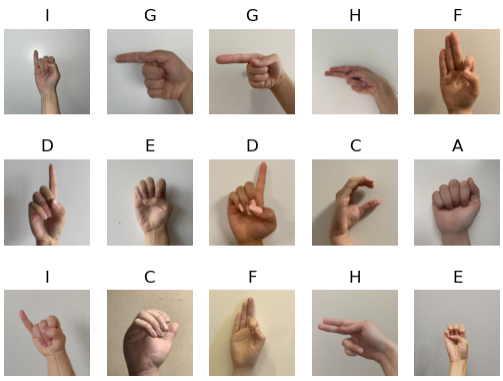
For loading the data, you can use plt.imread as in Lab 1, or any other method that you choose. You may find torchvision.datasets.ImageFolder helpful. (see https://pytorch.org/docs/stable/torchvision/datasets.html?highlight=image%20folder#torchvision.datasets.ImageFolder)

In [2]:
```
import numpy as np
import time
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
from torch.utils.data.sampler import SubsetRandomSampler
import torchvision.transforms as transforms

from torch.utils.data import random_split
```

In [15]:
```
classes = ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I')
```

```
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.

# 60% training, 20% validation and 20% testing samples split using "random_split"
images_dataset = torchvision.datasets.ImageFolder("./Lab3_Gestures_Summer/", transf
train_dataset, val_dataset, test_dataset = random_split(images_dataset, [0.6, 0.2,

print(len(train_dataset), len(val_dataset), len(test_dataset))

# Prepare DataLoader
batch_size = 30
num_workers = 1

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)

val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)

test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)

import matplotlib.pyplot as plt

# Looking at first image of each batch
k = 0
for images, labels in train_loader:
    image = images[0]
    # place the colour channel at the end, instead of at the beginning
    img = np.transpose(image, [1,2,0])
    # normalize pixel intensity values to [0, 1]
    img = img / 2 + 0.5
    plt.subplot(3, 5, k+1)
    plt.axis('off')
    plt.title(classes[labels[0]])
    plt.imshow(img)

    k += 1
    if k > 14:
        break
```

```
1332 444 443
```



2. Model Building and Sanity Checking [15 pt]

Part (a) Convolutional Network - 5 pt

Build a convolutional neural network model that takes the (224x224 RGB) image as input, and predicts the gesture letter. Your model should be a subclass of nn.Module. Explain your choice of neural network architecture: how many layers did you choose? What types of layers did you use? Were they fully-connected or convolutional? What about other decisions like pooling layers, activation functions, number of channels / hidden units?

In [14]:
```
torch.manual_seed(10) # set the random seed
from math import floor

class CNNet(nn.Module):
    def __init__(self, name="CNNet"):
        super(CNNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 5, 3)
        self.conv2 = nn.Conv2d(5, 10, 5)
        self.conv3 = nn.Conv2d(10, 25, 8)

        self.pool = nn.MaxPool2d(2, 2)

        # Computing the correct input size into the Fully Connected Layer
        self.a = floor((224 - 3 + 1)/2)
        self.b = floor((self.a - 5 + 1)/2)
        self.c = floor((self.b - 8 + 1)/2)

        self.fc1 = nn.Linear( 25*self.c*self.c , 32)
        self.fc2 = nn.Linear(32, 9)
```

```
        self.name = name

    def forward(self, img):
        x = self.pool(F.relu(self.conv1(img)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 25*self.c*self.c)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Part (b) Training Code - 5 pt

Write code that trains your neural network given some training data. Your training code should make it easy to tweak the usual hyperparameters, like batch size, learning rate, and the model object itself. Make sure that you are checkpointing your models from time to time (the frequency is up to you). Explain your choice of loss function and optimizer.

In [32]:
```
def get_model_name(name, batch_size, learning_rate, epoch):
    path = "model_{0}_bs{1}_lr{2}_epoch{3}".format(name,
                                                   batch_size,
                                                   learning_rate,
                                                   epoch)
    return path

def accuracy(net, data_loader):
    correct = 0
    total = 0
    for inputs, labels in data_loader:
        inputs = inputs.cuda()
        labels = labels.cuda()

        output = net(inputs)

        #select index with maximum prediction score
        pred = output.max(1, keepdim=True)[1]
        correct += pred.eq(labels.view_as(pred)).sum().item()
        total += inputs.shape[0]
    return correct / total

def train_net(net, batch_size=64, learning_rate=0.01, num_epochs=30):
    torch.manual_seed(1000)

    # Prepare DataLoader
    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size

    val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size, sh

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.parameters(), lr=learning_rate, momentum=0.9)

    train_accuracy = np.zeros(num_epochs)
    val_accuracy = np.zeros(num_epochs)
```

```
    for epoch in range(num_epochs):  # Loop over the dataset multiple times
        for i, data in enumerate(train_loader, 0):
            # Get the inputs
            inputs, labels = data
            inputs = inputs.cuda()
            labels = labels.cuda()
            # Zero the parameter gradients
            optimizer.zero_grad()
            # Forward pass, backward pass, and optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            # Calculate the statistics

        train_accuracy[epoch] = accuracy(net, train_loader)
        val_accuracy[epoch] = accuracy(net, val_loader)
        print(("Epoch {}: Train accuracy: {} | Validation accuracy: {}")
              .format(
                    epoch + 1,
                    accuracy(net, train_loader),
                    accuracy(net, val_loader)))
        # Save the current model (checkpoint) to a file
        model_path = get_model_name(net.name, batch_size, learning_rate, epoch)
        torch.save(net.state_dict(), "./testing/"+model_path)
    print('Finished Training')

    return train_accuracy, val_accuracy, num_epochs
```

Part (c) "Overfit" to a Small Dataset - 5 pt

One way to sanity check our neural network model and training code is to check whether the model is capable of "overfitting" or "memorizing" a small dataset. A properly constructed CNN with correct training code should be able to memorize the answers to a small number of images quickly.

Construct a small dataset (e.g. just the images that you have collected). Then show that your model and training code is capable of memorizing the labels of this small data set.

With a large batch size (e.g. the entire small dataset) and learning rate that is not too high, You should be able to obtain a 100% training accuracy on that small dataset relatively quickly (within 200 iterations).

In [39]:
```
def overtrain_net(net, small_loader, learning_rate=0.01, num_epochs=30):
    torch.manual_seed(1000)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.parameters(), lr=learning_rate, momentum=0.9)

    for epoch in range(num_epochs):  # Loop over the dataset multiple times
        for i, data in enumerate(small_loader, 0):
```

```
            # Get the inputs
            inputs, labels = data
            inputs = inputs.cuda()
            labels = labels.cuda()
            # Zero the parameter gradients
            optimizer.zero_grad()
            # Forward pass, backward pass, and optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            # Calculate the statistics

        print(("Epoch {}: Train accuracy: {}")
              .format(epoch + 1, accuracy(net, small_loader)))
    print('Finished Training')
```

In [41]:
```
classes = ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I')

transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.

images_dataset = torchvision.datasets.ImageFolder("./Lab3_Gestures_Summer/", transf
small_dataset = random_split(images_dataset, [0.02, 0.98])

print(len(small_dataset[0]))

# Prepare DataLoader
batch_size = 45

small_loader = torch.utils.data.DataLoader(small_dataset[0], batch_size=batch_size,

# Train the Model and try to overfit
CNN = CNNet()
CNN.cuda()

overtrain_net(CNN, small_loader, 0.01, num_epochs=120)
```

3. Hyperparameter Search [10 pt]

Part (a) - 1 pt

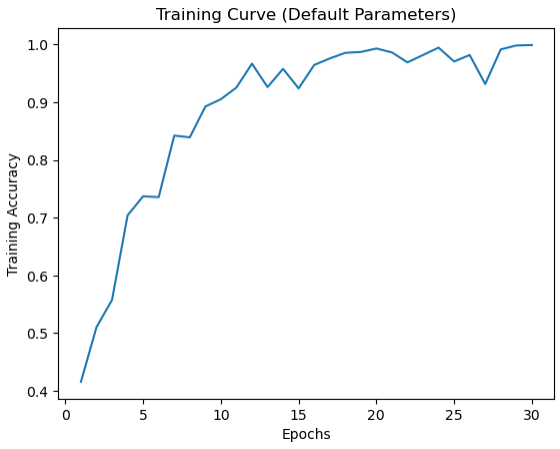
List 3 hyperparameters that you think are most worth tuning. Choose at least one hyperparameter related to the model architecture.

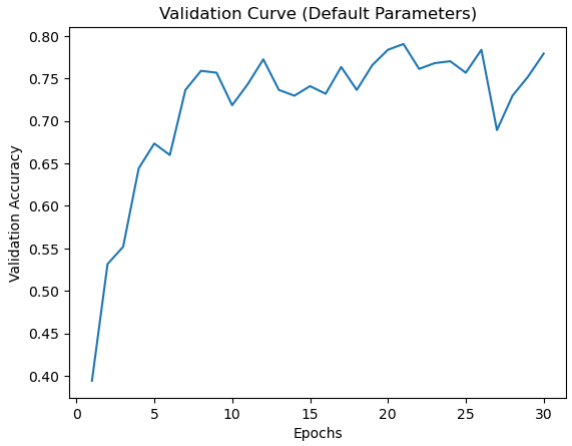
In [ ]:
```
# batch size
# the kernel size in each convolutional layer
# num of epochs
```

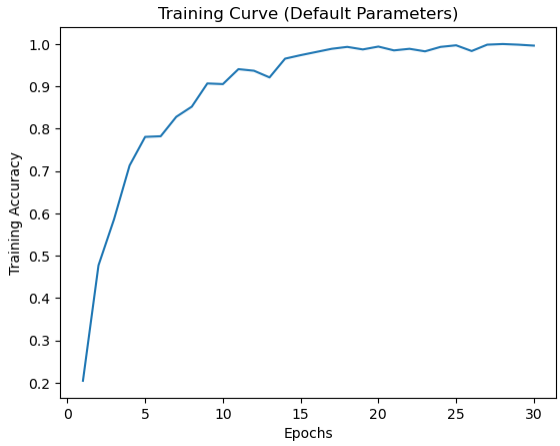
Part (b) - 5 pt

Tune the hyperparameters you listed in Part (a), trying as many values as you need to until you feel satisfied that you are getting a good model. Plot the training curve of at least 4 different hyperparameter settings.

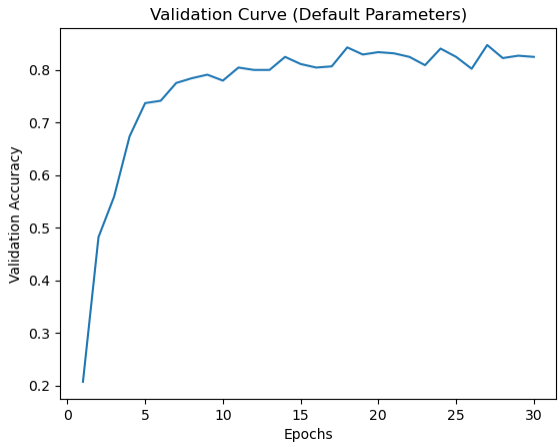
In [9]:
```
# First, default settings
CNN = CNNet()
CNN.cuda()

train_acc, val_acc, num_epoches = train_net(CNN)

plt.plot(np.arange(1, num_epoches + 1), train_acc)
plt.title("Training Curve (Default Parameters)")
plt.xlabel("Epochs")
plt.ylabel("Training Accuracy")
plt.show()

plt.plot(np.arange(1, num_epoches + 1), val_acc)
plt.title("Validation Curve (Default Parameters)")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.show()
```







In [12]:
```
# Next, adjust kernal sizes from (10, 5, 3) to (3, 5, 8)
CNN = CNNet("CNNet_Ker")
CNN.cuda()

train_acc, val_acc, num_epoches = train_net(CNN)

plt.plot(np.arange(1, num_epoches + 1), train_acc)
plt.title("Training Curve (Default Parameters)")
plt.xlabel("Epochs")
plt.ylabel("Training Accuracy")
plt.show()

plt.plot(np.arange(1, num_epoches + 1), val_acc)
plt.title("Validation Curve (Default Parameters)")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.show()
```



Part (d) - 2 pt

Report the test accuracy of your best model. You should only do this step once and prior to this step you should have only used the training and validation data.

In [42]:
```
CNN = CNNet(name = "CNNet_Ker")
CNN.cuda()
model_path = get_model_name(CNN.name, batch_size=64, learning_rate=0.01, epoch=18)
state = torch.load("./testing/" + model_path)
CNN.load_state_dict(state)

test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=True
accuracy(CNN, test_loader)
```

Out[42]: 0.9322799097065463

4. Transfer Learning [15 pt]

For many image classification tasks, it is generally not a good idea to train a very large deep neural network model from scratch due to the enormous compute requirements and lack of sufficient amounts of training data.

One of the better options is to try using an existing model that performs a similar task to the one you need to solve. This method of utilizing a pre-trained network for other similar tasks is broadly termed **Transfer Learning**. In this assignment, we will use Transfer Learning to extract features from the hand gesture images. Then, train a smaller network to use these features as input and classify the hand gestures.

As you have learned from the CNN lecture, convolution layers extract various features from the images which get utilized by the fully connected layers for correct classification. AlexNet architecture played a pivotal role in establishing Deep Neural Nets as a go-to tool for image classification problems and we will use an ImageNet pre-trained AlexNet model to extract features in this assignment.

Part (a) - 5 pt

Here is the code to load the AlexNet network, with pretrained weights. When you first run the code, PyTorch will download the pretrained weights from the internet.

In [21]:
```
import torchvision.models
alexnet = torchvision.models.alexnet(pretrained=True)
```

The alexnet model is split up into two components: *alexnet.features* and *alexnet.classifier*. The first neural network component, *alexnet.features*, is used to compute convolutional features, which are taken as input in *alexnet.classifier*.

The neural network alexnet.features expects an image tensor of shape Nx3x224x224 as input and it will output a tensor of shape Nx256x6x6 . (N = batch size).

Compute the AlexNet features for each of your training, validation, and test data. Here is an example code snippet showing how you can compute the AlexNet features for some images (your actual code might be different):

In [ ]:
```
# img = ... a PyTorch tensor with shape [N,3,224,224] containing hand images ...
features = alexnet.features(img)
```

In [64]:
```
import os

# Prepare DataLoader
batch_size = 1
num_workers = 1

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)

val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)

test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)

n = 0
for img, label in train_loader:
  features = alexnet.features(img)
  features_tensor = torch.from_numpy(features.detach().numpy())
  folder_name = './alex/train/' + str(classes[label])
  if not os.path.isdir(folder_name):
    os.mkdir(folder_name)
  torch.save(features_tensor.squeeze(0), folder_name + '/' + str(n) + '.tensor')
  n += 1

n = 0
for img, label in val_loader:
  features = alexnet.features(img)
  features_tensor = torch.from_numpy(features.detach().numpy())

  folder_name = './alex/val/' + str(classes[label])
  if not os.path.isdir(folder_name):
    os.mkdir(folder_name)
  torch.save(features_tensor.squeeze(0), folder_name + '/' + str(n) + '.tensor')
  n += 1

n = 0
for img, label in test_loader:
  features = alexnet.features(img)
  features_tensor = torch.from_numpy(features.detach().numpy())

  folder_name = './alex/test/' + str(classes[label])
  if not os.path.isdir(folder_name):
    os.mkdir(folder_name)
  torch.save(features_tensor.squeeze(0), folder_name + '/' + str(n) + '.tensor')
  n += 1
```

Part (b) - 3 pt

Build a convolutional neural network model that takes as input these AlexNet features, and makes a prediction. Your model should be a subclass of nn.Module.

Explain your choice of neural network architecture: how many layers did you choose? What types of layers did you use: fully-connected or convolutional? What about other decisions like pooling layers, activation functions, number of channels / hidden units in each layer?

Here is an example of how your model may be called:

In [65]:
```
train_features = torchvision.datasets.DatasetFolder('./alex/train', loader=torch.lo
val_features = torchvision.datasets.DatasetFolder('./alex/val', loader=torch.load,
test_features = torchvision.datasets.DatasetFolder('./alex/test', loader=torch.load

train_feature_loader = torch.utils.data.DataLoader(train_features, batch_size=64, n
val_feature_loader = torch.utils.data.DataLoader(val_features, batch_size=64, num_w
test_feature_loader = torch.utils.data.DataLoader(test_features, batch_size=64, num
```

In [66]:
```
torch.manual_seed(10) # set the random seed
from math import floor

class AlexNet(nn.Module):
    def __init__(self, name="AlexNet"):
        super(AlexNet, self).__init__()
        self.conv1 = nn.Conv2d(256, 512, 3)
        self.pool = nn.MaxPool2d(2, 2)

        # Computing the correct input size into the Fully Connected Layer
        self.x = floor((6 - 3 + 1)/2)
        self.FC_input = 512*self.x*self.x

        self.fc1 = nn.Linear(self.FC_input, 32)
        self.fc2 = nn.Linear(32, 9)

        self.name = name

    def forward(self, features):
        x = self.pool(F.relu(self.conv1(features)))
        x = x.view(-1, self.FC_input)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.softmax(x, dim=1)
```

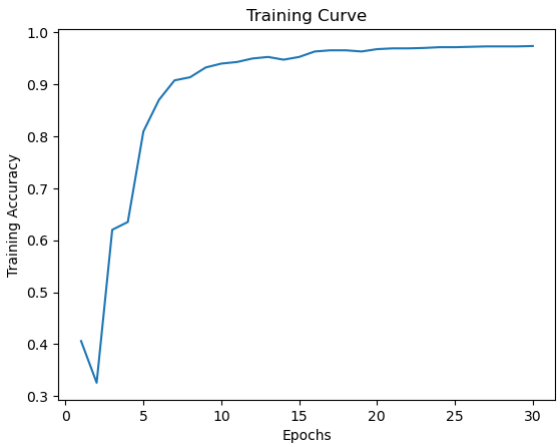
Part (c) - 5 pt

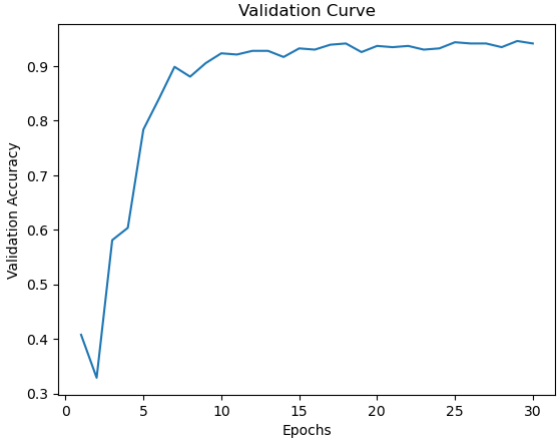
Train your new network, including any hyperparameter tuning. Plot and submit the training curve of your best model only.

Note: Depending on how you are caching (saving) your AlexNet features, PyTorch might still be tracking updates to the **AlexNet weights**, which we are not tuning. One workaround is to convert your AlexNet feature tensor into a numpy array, and then back into a PyTorch tensor.

In [70]:
```
def train_alex(net, batch_size=64, learning_rate=0.01, num_epochs=30):
    torch.manual_seed(1000)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.parameters(), lr=learning_rate, momentum=0.9)

    train_accuracy = np.zeros(num_epochs)
    val_accuracy = np.zeros(num_epochs)

    for epoch in range(num_epochs):  # Loop over the dataset multiple times
        for inputs, labels in iter(train_feature_loader):
            inputs = inputs.cuda()
            labels = labels.cuda()
            # Zero the parameter gradients
            optimizer.zero_grad()
            # Forward pass, backward pass, and optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            # Calculate the statistics

        train_accuracy[epoch] = accuracy(net, train_feature_loader)
        val_accuracy[epoch] = accuracy(net, val_feature_loader)
        print(("Epoch {}: Train accuracy: {} | Validation accuracy: {}")
              .format(
                    epoch + 1,
                    accuracy(net, train_feature_loader),
                    accuracy(net, val_feature_loader)))
        # Save the current model (checkpoint) to a file
        model_path = get_model_name(net.name, batch_size, learning_rate, epoch)
        torch.save(net.state_dict(), "./testing/"+model_path)
    print('Finished Training')

    return train_accuracy, val_accuracy, num_epochs
```

In [71]:
```
ANN = AlexNet()
ANN.cuda()

training_accuracy, validation_accuracy, num_epoches = train_alex(ANN)

plt.plot(np.arange(1, num_epoches + 1), training_accuracy)
plt.title("Training Curve")
plt.xlabel("Epochs")
plt.ylabel("Training Accuracy")
plt.show()

plt.plot(np.arange(1, num_epoches + 1), validation_accuracy)
plt.title("Validation Curve")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
```





Part (d) - 2 pt

Report the test accuracy of your best model. How does the test accuracy compare to Part 3(d) without transfer learning?

In [75]:
```
AN = AlexNet()
AN.cuda()
model_path = get_model_name(AN.name, batch_size=64, learning_rate=0.01, epoch=30-1)
state = torch.load("./testing/" + model_path)
AN.load_state_dict(state)

accuracy(AN, test_feature_loader)
```

Out[75]: 0.9300225733634312

5. Additional Testing [5 pt]

As a final step in testing we will be revisiting the sample images that you had collected and submitted at the start of this lab. These sample images should be untouched and will be used to demonstrate how well your model works at identifying your hand guestures.

Using the best transfer learning model developed in Part 4. Report the test accuracy on your sample images and how it compares to the test accuracy obtained in Part 4(d)? How well did your model do for the different hand guestures? Provide an explanation for why you think your model performed the way it did?

In [ ]:
```
accuracy(AN, test_loader)
```