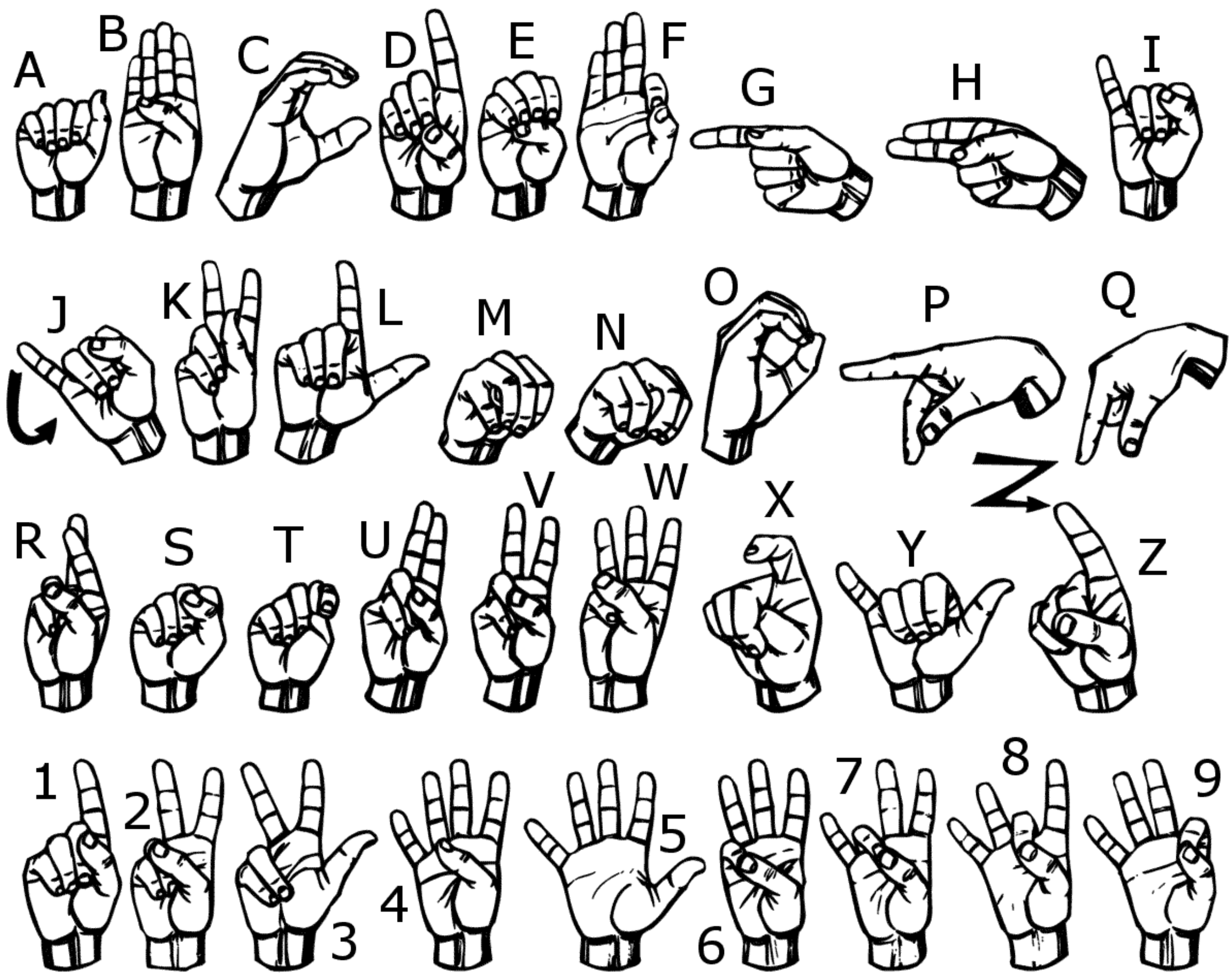
American Sign Language

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 alphabets are shown below. This excercise 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.



Data Loading

The data for this excercise is present in "[asl_data.zip (https://www.dropbox.com/s/r75maq5e1vyda4g/asl_data.zip?dl=0)](https://www.dropbox.com/s/r75maq5e1vyda4g/asl_data.zip?dl=0)". The dataset contains 9 classes (images corresponding to characters A to I). For convenience, the dataset is structured in such a way that we can use TorchVision's ImageFolder dataset [(documentation) (https://pytorch.org/docs/stable/torchvision/datasets.html#torchvision.datasets.ImageFolder)](https://pytorch.org/docs/stable/torchvision/datasets.html#torchvision.datasets.ImageFolder) rather than writing your own custom dataset loader.

In [61]:
```
# Define the standard imports
from __future__ import print_function
from __future__ import division

import torch
import torch.nn as nn
import torch.optim as optim

import numpy as np

import torchvision
from torchvision import datasets, models, transforms

import matplotlib.pyplot as plt
%matplotlib inline

import time
import os
import copy

#%env CUDA_VISIBLE_DEVICES=2
import torch, torchvision
from torchvision import datasets, models, transforms
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
import time
from torchsummary import summary

import numpy as np
import matplotlib.pyplot as plt
import os

from PIL import Image
```

In [ ]:
```
# Download the data in the current working directory
!rm -rf asl_data.zip asl_data
!wget -O asl_data.zip https://www.dropbox.com/s/r75maq5e1vyda4g/asl_data.zip?d
l=0
!unzip asl_data.zip
!rm asl_data.zip

# Top level data directory. Here we assume the format of the directory conform
s
#   to the ImageFolder structure
data_dir = "./asl_data"

# Define the class label
class_dict = {0:'A', 1:'B', 2:'C', 3:'D', 4:'E', 5:'F', 6:'G', 7:'H', 8:'I'}
```

In [ ]:
```
len(class_dict)
```

Out[ ]: 9

Visualize the data

We will now see how the sample data looks like

In [3]:
```
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), transform
=transforms.ToTensor()) for x in ['train', 'val']}
val_dataloader = torch.utils.data.DataLoader(image_datasets['val'], batch_size
=1, shuffle=True)


f = plt.figure(figsize=(10, 10))
for i in range(18):
    img, label = next(iter(val_dataloader))
    img = img.squeeze().permute(1,2,0).numpy()

    plt.subplot(3,6,i+1)
    plt.imshow(img)
    plt.xlabel(class_dict[label.numpy()[0]])
```

Excercise: Neural Network

In this excercise you will be using a neural network. You are free to use one of the pretrained model, as demonstrated in the previous lab, or write your own neural network from scratch.

You may use the PyTorch documentation, previous excercises and notebooks freely. You might find documentations and notebooks discussed in the last two classes helpful. However, all code and analysis that you submit must be your own.

Questions

Question 1: Model Building

Build a multi-layered perceptron (MLP) in Pytorch that inputs that takes the (224x224 RGB) image as input, and predicts the letter (You may need to flatten the image vector first). Your model should be a subclass of `nn.Module`. Explain your choice of neural network architecture: how many layers your network has? What types of layers does it contain? What about other decisions like use of dropout layers, activation functions, number of channels / hidden units.

Question 2: Training Code

Write code to train your neural network given some training data. Your training code should make it easy to tweak hyperparameters. Make sure that you are checkpointing your models from time to time (the frequency is up to you). Explain your choice of loss function. Ensure that your code runs on GPU.

Question 3: Overfit to a Small Dataset

Part (a): One way to sanity check our neural network model and training code is to check whether the model is capable of overfitting a small dataset. Construct a small dataset (e.g. 1-2 image per class). Then show that your model and training code is capable of overfitting on that small dataset. You should be able to obtain a 100% training accuracy on that small dataset relatively quickly.

If your model cannot overfit the small dataset quickly, then there is a bug in either your model code and/or your training code. Fix the issues before you proceed to the next step.

Part (b): Once you are done with the above part, try to reduce the effect of overfitting by using techniques discussed in the previous lecture.

Question 4: Finetuning

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.

In this part, you will use Transfer Learning to extract features from the hand gesture images. Then, train last few classification layers to use these features as input and classify the hand gestures. As you have learned in the previous lecture, you can use AlexNet architecture that is pretrained on 1000-class ImageNet dataset and finetune it for the task of understanding American sign language.

Question 5: Report result

Train your new network, including any hyperparameter tuning. Plot and submit the training and validation loss and accuracy of your best model only. Along with it, also submit the final validation accuracy achieved by your model.

In [4]:
```
BATCH_SIZE = 64

trainloader = torch.utils.data.DataLoader(image_datasets['train'], batch_size=BATCH_SIZE, shuffle=True)
testloader = torch.utils.data.DataLoader(image_datasets['val'], batch_size=BATCH_SIZE, shuffle=True)

print('Number of training images: {}'.format(len(trainloader)))
print('Number of validation images: {}'.format(len(testloader)))
```

```
Number of training images: 15
Number of validation images: 4
```

Question 1: Model Building

In [5]:
```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.layer1 = nn.Linear(224 * 224 * 3, 50)
        self.layer2 = nn.Linear(50, 20)
        self.layer3 = nn.Linear(20, 9)
    def forward(self, img):
        flattened = img.view(-1, 224 * 224 * 3)
        activation1 = F.relu(self.layer1(flattened))
        activation2 = F.relu(self.layer2(activation1))
        output = self.layer3(activation2)
        return output
```

In [62]:
```
model = Net()
# Ship data and model to GPU if available
device = "cuda" if torch.cuda.is_available() else "cpu"
model = model.to(device)
```

In [8]:
```
Num_Params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print('Number of parameters ' ,Num_Params)
```

```
Number of parameters  7527659
```

Our model is an MLP with

1. Input layer of size 224 *224* 3 size
2. Two hidden layers of 250 and 100 nodes
3. Output layer of 10 nodes
4. Since RGB image , number of channels are set to 3
5. ReLU activaion function is used
6. Number of parameters of the MLP = 7527659

Question 2: Training Code

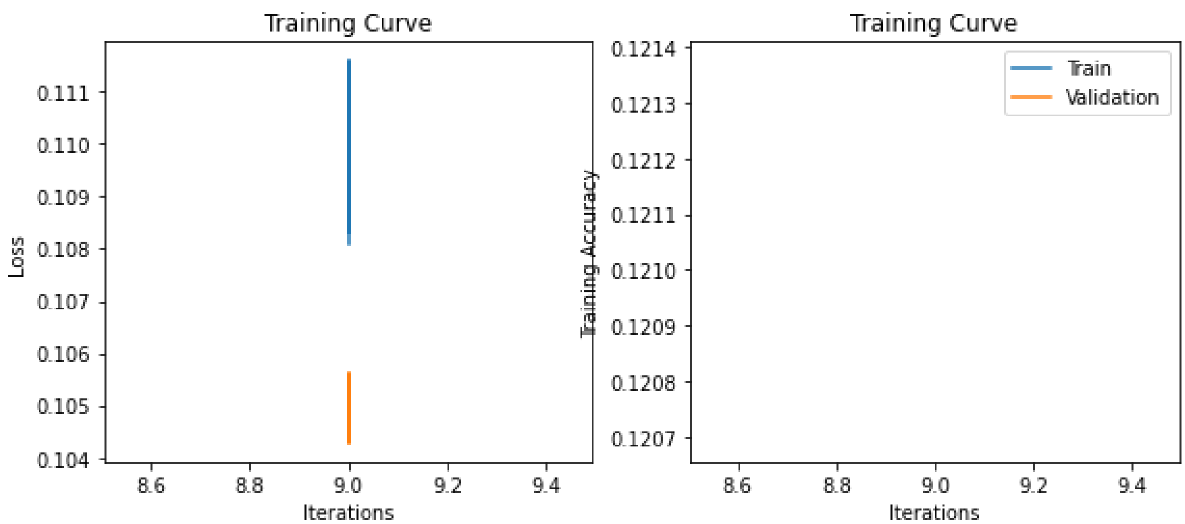
In [9]:
```
# Ship data and model to GPU if available
device = "cuda" if torch.cuda.is_available() else "cpu"
model = model.to(device)
```

Model is set o CUDA

In [64]:
```
def trainval(model, train_data, valid_data, device, batch_size=20, num_iters=1
, learn_rate=0.01, weight_decay=0):
    train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_si
ze, shuffle=True) # shuffle after every epoch
    val_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size
)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=learn_rate, momentum=0.9, wei
ght_decay=weight_decay)

    iters, losses, val_losses, train_acc, val_acc = [], [], [], [], []

    # training
    n = 0 # the number of iterations
    for n in range(num_iters):
        for imgs, labels in train_loader:
            imgs, labels = imgs.to(device), labels.to(device)

            model.train() #*****************************#
            optimizer.zero_grad()         # a clean up step for PyTorch
            out = model(imgs)             # forward pass
            loss = criterion(out, labels) # compute the total loss
            loss.backward()               # backward pass (compute parameter u
pdates)
            optimizer.step()              # make the updates for each paramete
r

            # save the current training information
            if n % 10 == 9:
                iters.append(n)
                losses.append(float(loss)/batch_size)        # compute *averag
e* loss
                train_accuracy = get_accuracy(model, train_data, device)
                val_accuracy = get_accuracy(model, valid_data, device)
                for im, lb in val_loader:
                    im, lb = im.to(device), lb.to(device)
                    val_out = model(im)
                    val_loss = criterion(val_out, lb)
                val_losses.append(float(val_loss)/batch_size)
                train_acc.append(train_accuracy) # compute training accuracy
                val_acc.append(val_accuracy)   # compute validation accuracy

    # plotting
    plt.figure(figsize=(10,4))
    plt.subplot(1,2,1)
    plt.title("Training Curve")
    plt.plot(iters, losses, label="Train")
    plt.plot(iters, val_losses, label="Validation")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")

    plt.subplot(1,2,2)
    plt.title("Training Curve")
    plt.plot(iters, train_acc, label="Train")
    plt.plot(iters, val_acc, label="Validation")
```

```
    plt.xlabel("Iterations")
    plt.ylabel("Training Accuracy")
    plt.legend(loc='best')
    plt.show()

    print("Final Training Accuracy: {}".format(train_acc[-1]))
    print("Final Validation Accuracy: {}".format(val_acc[-1]))


def get_accuracy(model, data, device):
    correct = 0
    total = 0

    model.eval() #*********#
    for imgs, labels in torch.utils.data.DataLoader(data, batch_size=64):
        imgs, labels = imgs.to(device), labels.to(device)
        output = model(imgs)
        pred = output.max(1, keepdim=True)[1] # get the index of the max logit
        correct += pred.eq(labels.view_as(pred)).sum().item()
        total += imgs.shape[0]
        accuracy = correct / total
    return accuracy
```

In [34]:
```
trainval(model, image_datasets['train'], image_datasets['val'], device, num_it
ers=10)
```




```
Final Training Accuracy: [0.12137486573576799, 0.12137486573576799, 0.1213748
6573576799, 0.12137486573576799, 0.12137486573576799, 0.12137486573576799, 0.
12137486573576799, 0.12137486573576799, 0.12137486573576799, 0.12137486573576
799, 0.12137486573576799, 0.12137486573576799, 0.12137486573576799, 0.1213748
6573576799, 0.12137486573576799, 0.12137486573576799, 0.12137486573576799, 0.
12137486573576799, 0.12137486573576799, 0.12137486573576799, 0.12137486573576
799, 0.12137486573576799, 0.12137486573576799, 0.12137486573576799, 0.1213748
6573576799, 0.12137486573576799, 0.12137486573576799, 0.12137486573576799, 0.
12137486573576799, 0.12137486573576799, 0.12137486573576799, 0.12137486573576
799, 0.12137486573576799, 0.12137486573576799, 0.12137486573576799, 0.1213748
6573576799, 0.12137486573576799, 0.12137486573576799, 0.12137486573576799, 0.
12137486573576799, 0.12137486573576799, 0.12137486573576799, 0.12137486573576
799, 0.12137486573576799, 0.12137486573576799, 0.12137486573576799]
Final Validation Accuracy: [0.1206896551724138, 0.1206896551724138, 0.1206896
551724138, 0.1206896551724138, 0.1206896551724138, 0.1206896551724138, 0.1206
896551724138, 0.1206896551724138, 0.1206896551724138, 0.1206896551724138, 0.1
206896551724138, 0.1206896551724138, 0.1206896551724138, 0.1206896551724138,
0.1206896551724138, 0.1206896551724138, 0.1206896551724138, 0.120689655172413
8, 0.1206896551724138, 0.1206896551724138, 0.1206896551724138, 0.120689655172
4138, 0.1206896551724138, 0.1206896551724138, 0.1206896551724138, 0.120689655
1724138, 0.1206896551724138, 0.1206896551724138, 0.1206896551724138, 0.120689
6551724138, 0.1206896551724138, 0.1206896551724138, 0.1206896551724138, 0.120
6896551724138, 0.1206896551724138, 0.1206896551724138, 0.1206896551724138, 0.
1206896551724138, 0.1206896551724138, 0.1206896551724138, 0.1206896551724138,
0.1206896551724138, 0.1206896551724138, 0.1206896551724138, 0.120689655172413
8, 0.1206896551724138]
```

1. Hyperparameters like like epochs,leraning rate,batch_size,weight_decay are included
2. Checkpointing after avery 10 iterations
3. Device is CUDA
4. Loss function is chosen is CrossEntropyLoss() as this is multiclass classfication.It computes the *softmax* activation function on the supplied predictions as well as the actual loss via *negative log likelihood.*

Question 3: Overfit to a Small Dataset

Part(a)

1. I have created overfitting condition by Dropping few images per class
2. It is noticed Training acc is 11% and Val acc is 1%

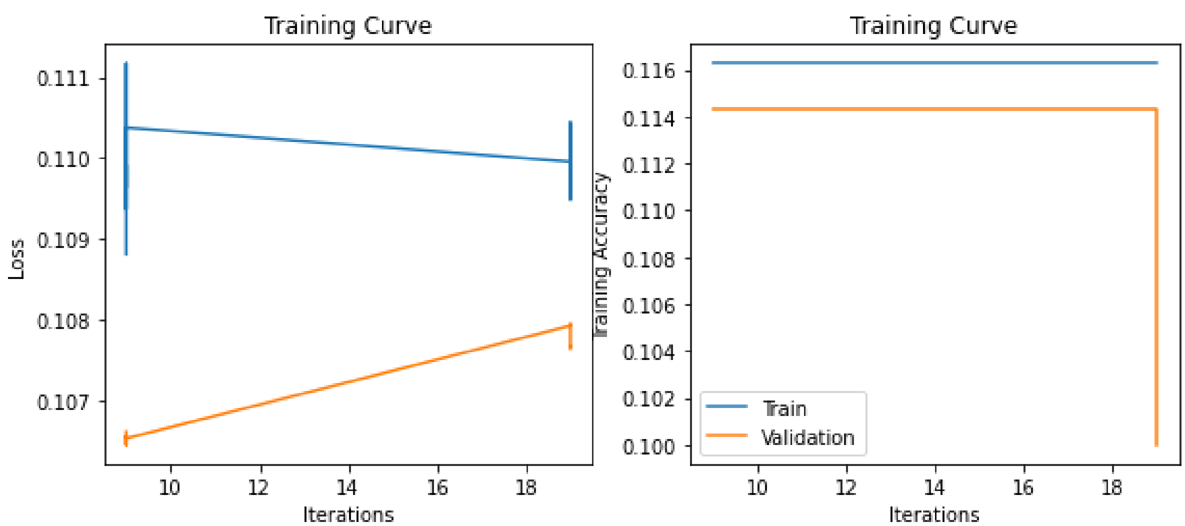
In [103]:
```
!rm -r /content/asl_data2
```

In [104]:
```
!cp -r asl_data asl_data2
```

In [ ]:
```
!find /content/asl_data2/train -depth -path "*0*.jpg"
```

In [119]:
```
!find /content/asl_data2/val/ -depth -path '*0*' -delete
!find /content/asl_data2/train/ -depth -path '*0*' -delete
```

In [120]:
```
data_dir2='/content/asl_data2'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir2, x), transfor
m=transforms.ToTensor()) for x in ['train', 'val']}

trainval(model, image_datasets['train'], image_datasets['val'], device, num_it
ers=20)
```




```
Final Training Accuracy: 0.11627906976744186
Final Validation Accuracy: 0.1
```

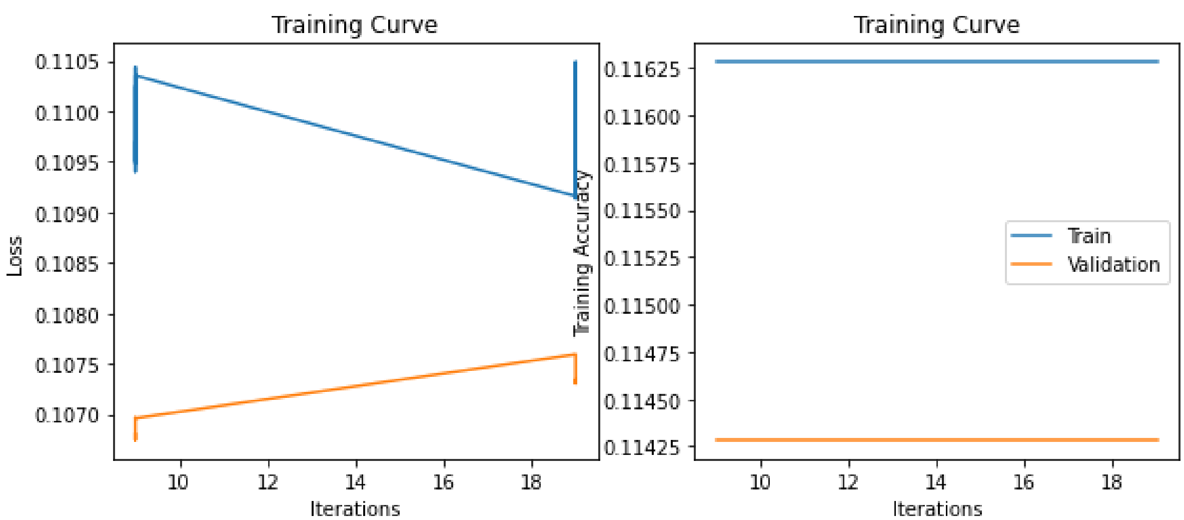
In [75]:
```
image_datasets
```

Out[75]:
```
{'train': Dataset ImageFolder
     Number of datapoints: 314
     Root location: /content/asl_data2/train
     StandardTransform
 Transform: ToTensor(), 'val': Dataset ImageFolder
     Number of datapoints: 79
     Root location: /content/asl_data2/val
     StandardTransform
 Transform: ToTensor()}
```

Part(b)

1. Used Dropout technique to reduce overfit condition simulated above
2. Both Train and Val accuracies have become 11%

In [121]:
```
class NetrWithDropout(nn.Module):
    def __init__(self):
        super(NetrWithDropout, self).__init__()
        self.layer1 = nn.Linear(224 * 224 * 3, 50)
        self.layer2 = nn.Linear(50, 20)
        self.layer3 = nn.Linear(20, 9)
        self.dropout1 = nn.Dropout(0.4) # drop out layer with 40% dropped out
 neuron
        self.dropout2 = nn.Dropout(0.4)
        self.dropout3 = nn.Dropout(0.4)
    def forward(self, img):
        flattened = img.view(-1, 224 * 224 * 3)
        activation1 = F.relu(self.layer1(flattened))
        activation2 = F.relu(self.layer2(activation1))
        output = self.layer3(activation2)
        return output

data_dir2='/content/asl_data2'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir2, x), transfor
m=transforms.ToTensor()) for x in ['train', 'val']}

model = NetrWithDropout()
# Ship data and model to GPU if available
device = "cuda" if torch.cuda.is_available() else "cpu"
model = model.to(device)

trainval(model, image_datasets['train'], image_datasets['val'], device, num_it
ers=20)
```



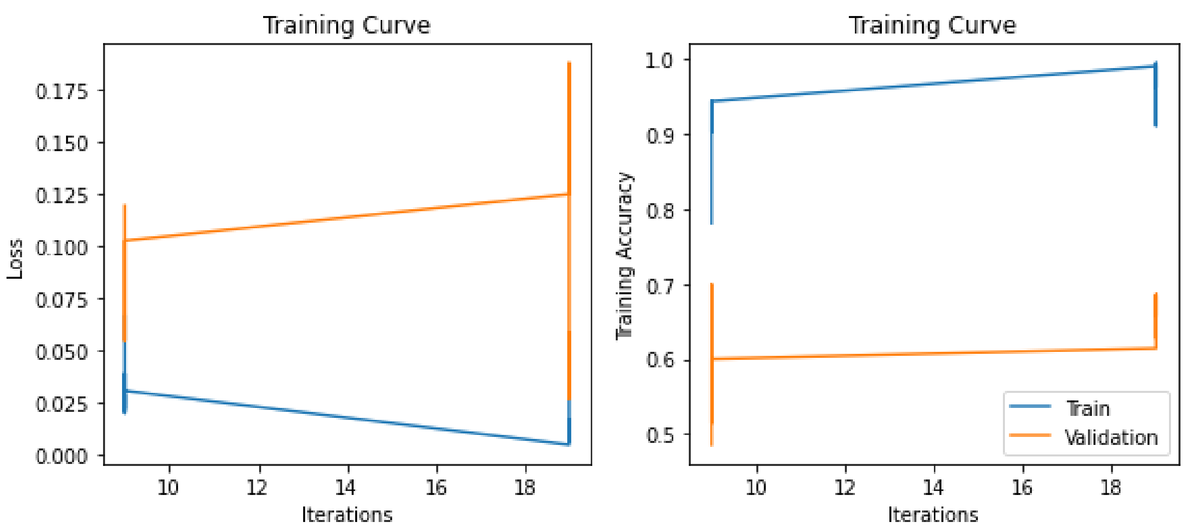

```
Final Training Accuracy: 0.11627906976744186
Final Validation Accuracy: 0.11428571428571428
```

Question 4: Finetuning

1. Finteuned with **Resnet50** for Transfer learning
2. Accuracy is singinifincantly than MLP implementation

In [122]:
```
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), transform
=transforms.ToTensor()) for x in ['train', 'val']}
```

In [123]:
```
model = models.resnet50(pretrained=True)
for param in model.parameters():
    param.requires_grad = False

model.fc = nn.Sequential(nn.Linear(2048, 512),
                                 nn.ReLU(),
                                 nn.Dropout(0.2),
                                 nn.Linear(512, 9),
                                 nn.LogSoftmax(dim=1))
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=0.003)
model.to(device)
trainval(model, image_datasets['train'], image_datasets['val'], device, num_it
ers=20)
```




```
Final Training Accuracy: 0.9906976744186047
Final Validation Accuracy: 0.6285714285714286
```

Feature extration to reduce computation time using **Alexnet**

In [131]:
```
model = models.resnet50(pretrained=True)
# Models to choose from [resnet, alexnet, vgg, squeezenet, densenet, inceptio
n]
model_name = "alexnet"

# Number of classes in the dataset
num_classes = 9

# Batch size for training (change depending on how much memory you have)
batch_size = 8

# Number of epochs to train for
num_epochs = 15

# Flag for feature extracting. When False, we finetune the whole model,
#   when True we only update the reshaped layer params
feature_extract = True
```

In [132]:
```
def train_model(model, dataloaders, criterion, optimizer, num_epochs=25):
    since = time.time()

    val_acc_history = []

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

    for epoch in range(num_epochs):
        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':
                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]:
                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'):
                    # Get model outputs and calculate loss
                    outputs = model(inputs)
                    loss = criterion(outputs, labels)

                    _, preds = torch.max(outputs, 1)

                    # 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 / len(dataloaders[phase].dataset)
            epoch_acc = running_corrects.double() / len(dataloaders[phase].dat
aset)

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

            # deep copy the model
```

```
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())
            if phase == 'val':
                val_acc_history.append(epoch_acc)

        print()

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

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

In [133]:

```
def set_parameter_requires_grad(model, feature_extracting):
    if feature_extracting:
        for param in model.parameters():
            param.requires_grad = False
```

In [134]:
```
def initialize_model(model_name, num_classes, feature_extract, use_pretrained=
True):
    # Initialize these variables which will be set in this if statement. Each
 of these
    #   variables is model specific.
    model_ft = None
    input_size = 0

    """ Alexnet
    """
    model_ft = models.alexnet(pretrained=use_pretrained)
    set_parameter_requires_grad(model_ft, feature_extract)
    num_ftrs = model_ft.classifier[6].in_features
    model_ft.classifier[6] = nn.Linear(num_ftrs,num_classes)
    input_size = 224

    return model_ft, input_size

# Initialize the model for this run
model_ft, input_size = initialize_model(model_name, num_classes, feature_extra
ct, use_pretrained=True)

# Print the model we just instantiated
print(model_ft)
```

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=
False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=
False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode
=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=9, bias=True)
  )
)
```

In [135]:
```
# Data augmentation and normalization for training
# Just normalization for validation
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(input_size),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(input_size),
        transforms.CenterCrop(input_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}
data_dir='/content/asl_data'

print("Initializing Datasets and Dataloaders...")

# Create training and validation datasets
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), data_tran
sforms[x]) for x in ['train', 'val']}
# Create training and validation dataloaders
dataloaders_dict = {x: torch.utils.data.DataLoader(image_datasets[x], batch_si
ze=batch_size, shuffle=True, num_workers=4) for x in ['train', 'val']}

# Detect if we have a GPU available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
Initializing Datasets and Dataloaders...
```

In [136]:
```
# Send the model to GPU
model_ft = model_ft.to(device)

# Gather the parameters to be optimized/updated in this run. If we are
#  finetuning we will be updating all parameters. However, if we are
#  doing feature extract method, we will only update the parameters
#  that we have just initialized, i.e. the parameters with requires_grad
#  is True.
params_to_update = model_ft.parameters()
print("Params to learn:")
if feature_extract:
    params_to_update = []
    for name,param in model_ft.named_parameters():
        if param.requires_grad == True:
            params_to_update.append(param)
            print("\t",name)
else:
    for name,param in model_ft.named_parameters():
        if param.requires_grad == True:
            print("\t",name)

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

```
Params to learn:
         classifier.6.weight
         classifier.6.bias
```

In [141]:
```
# Setup the loss fxn
criterion = nn.CrossEntropyLoss()

num_epochs=100
# Train and evaluate
model_ft, hist = train_model(model_ft, dataloaders_dict, criterion, optimizer_ft, num_epochs=num_epochs)
```

```
Epoch 0/99
----------
train Loss: 1.1522 Acc: 0.6788
val Loss: 1.7535 Acc: 0.6422

Epoch 1/99
----------
train Loss: 1.3186 Acc: 0.6488
val Loss: 1.3802 Acc: 0.6422

Epoch 2/99
----------
train Loss: 1.3925 Acc: 0.6337
val Loss: 1.5144 Acc: 0.6853

Epoch 3/99
----------
train Loss: 1.2789 Acc: 0.6627
val Loss: 1.5857 Acc: 0.6552

Epoch 4/99
----------
train Loss: 1.2296 Acc: 0.6552
val Loss: 2.3207 Acc: 0.6164

Epoch 5/99
----------
train Loss: 1.4453 Acc: 0.6434
val Loss: 1.5395 Acc: 0.6853

Epoch 6/99
----------
train Loss: 1.2035 Acc: 0.6810
val Loss: 1.8049 Acc: 0.5991

Epoch 7/99
----------
train Loss: 1.2588 Acc: 0.6488
val Loss: 1.5399 Acc: 0.6767

Epoch 8/99
----------
train Loss: 1.0985 Acc: 0.6810
val Loss: 1.4666 Acc: 0.6983

Epoch 9/99
----------
train Loss: 1.1919 Acc: 0.6692
val Loss: 1.2711 Acc: 0.6810

Epoch 10/99
----------
train Loss: 1.2137 Acc: 0.6821
val Loss: 1.3413 Acc: 0.7241

Epoch 11/99
----------
```

```
train Loss: 1.1575 Acc: 0.6627
val Loss: 1.4555 Acc: 0.6552

Epoch 12/99
----------
train Loss: 1.2336 Acc: 0.6638
val Loss: 1.3025 Acc: 0.7155

Epoch 13/99
----------
train Loss: 1.1480 Acc: 0.6842
val Loss: 1.6078 Acc: 0.6164

Epoch 14/99
----------
train Loss: 1.2004 Acc: 0.6735
val Loss: 1.5320 Acc: 0.6293

Epoch 15/99
----------
train Loss: 1.2914 Acc: 0.6649
val Loss: 1.7646 Acc: 0.6250

Epoch 16/99
----------
train Loss: 1.2346 Acc: 0.6627
val Loss: 1.4854 Acc: 0.6810

Epoch 17/99
----------
train Loss: 1.2023 Acc: 0.6660
val Loss: 1.4062 Acc: 0.6724

Epoch 18/99
----------
train Loss: 1.1625 Acc: 0.6853
val Loss: 1.4687 Acc: 0.6422

Epoch 19/99
----------
train Loss: 1.2339 Acc: 0.6606
val Loss: 1.5862 Acc: 0.6681

Epoch 20/99
----------
train Loss: 1.0815 Acc: 0.7089
val Loss: 1.8706 Acc: 0.6466

Epoch 21/99
----------
train Loss: 1.2151 Acc: 0.6552
val Loss: 1.3631 Acc: 0.7026

Epoch 22/99
----------
train Loss: 1.1444 Acc: 0.6885
val Loss: 1.7008 Acc: 0.6509
```

```
Epoch 23/99
----------
train Loss: 1.2210 Acc: 0.6756
val Loss: 1.4725 Acc: 0.6681

Epoch 24/99
----------
train Loss: 1.2434 Acc: 0.6617
val Loss: 1.4232 Acc: 0.6509

Epoch 25/99
----------
train Loss: 1.2198 Acc: 0.6907
val Loss: 1.3863 Acc: 0.6853

Epoch 26/99
----------
train Loss: 1.6025 Acc: 0.6520
val Loss: 1.7072 Acc: 0.7112

Epoch 27/99
----------
train Loss: 1.2145 Acc: 0.6950
val Loss: 1.6406 Acc: 0.6293

Epoch 28/99
----------
train Loss: 1.1876 Acc: 0.6928
val Loss: 2.0145 Acc: 0.6034

Epoch 29/99
----------
train Loss: 1.2135 Acc: 0.6724
val Loss: 1.4185 Acc: 0.6853

Epoch 30/99
----------
train Loss: 1.0633 Acc: 0.7132
val Loss: 1.1617 Acc: 0.7112

Epoch 31/99
----------
train Loss: 1.1441 Acc: 0.6928
val Loss: 1.5182 Acc: 0.6767

Epoch 32/99
----------
train Loss: 1.1083 Acc: 0.7143
val Loss: 1.3470 Acc: 0.6897

Epoch 33/99
----------
train Loss: 1.1299 Acc: 0.6928
val Loss: 1.6757 Acc: 0.6595

Epoch 34/99
```

```
----------
train Loss: 1.1642 Acc: 0.6541
val Loss: 1.5313 Acc: 0.6724

Epoch 35/99
----------
train Loss: 1.1534 Acc: 0.6950
val Loss: 1.5914 Acc: 0.6379

Epoch 36/99
----------
train Loss: 1.2276 Acc: 0.6960
val Loss: 1.7598 Acc: 0.6121

Epoch 37/99
----------
train Loss: 1.1233 Acc: 0.7014
val Loss: 1.4835 Acc: 0.6681

Epoch 38/99
----------
train Loss: 1.2187 Acc: 0.6788
val Loss: 1.7479 Acc: 0.6681

Epoch 39/99
----------
train Loss: 1.1787 Acc: 0.6799
val Loss: 1.4104 Acc: 0.6552

Epoch 40/99
----------
train Loss: 1.2382 Acc: 0.6745
val Loss: 1.5420 Acc: 0.6767

Epoch 41/99
----------
train Loss: 1.0413 Acc: 0.7046
val Loss: 1.6236 Acc: 0.7069

Epoch 42/99
----------
train Loss: 1.1422 Acc: 0.6950
val Loss: 1.5426 Acc: 0.6897

Epoch 43/99
----------
train Loss: 1.2191 Acc: 0.6799
val Loss: 1.5290 Acc: 0.7026

Epoch 44/99
----------
train Loss: 1.0718 Acc: 0.6896
val Loss: 1.7210 Acc: 0.6379

Epoch 45/99
----------
train Loss: 1.2232 Acc: 0.6885
```

```
val Loss: 1.5475 Acc: 0.7026

Epoch 46/99
----------
train Loss: 1.1960 Acc: 0.6864
val Loss: 1.6552 Acc: 0.6595

Epoch 47/99
----------
train Loss: 1.1150 Acc: 0.7121
val Loss: 1.5715 Acc: 0.6767

Epoch 48/99
----------
train Loss: 1.1040 Acc: 0.7186
val Loss: 1.3890 Acc: 0.7155

Epoch 49/99
----------
train Loss: 1.2877 Acc: 0.6692
val Loss: 1.6720 Acc: 0.6595

Epoch 50/99
----------
train Loss: 1.0580 Acc: 0.6960
val Loss: 1.6301 Acc: 0.6897

Epoch 51/99
----------
train Loss: 1.0650 Acc: 0.7197
val Loss: 1.6735 Acc: 0.6853

Epoch 52/99
----------
train Loss: 0.9428 Acc: 0.7261
val Loss: 1.3650 Acc: 0.7414

Epoch 53/99
----------
train Loss: 1.1685 Acc: 0.6778
val Loss: 1.7959 Acc: 0.6681

Epoch 54/99
----------
train Loss: 1.3721 Acc: 0.6638
val Loss: 1.8587 Acc: 0.6940

Epoch 55/99
----------
train Loss: 1.0469 Acc: 0.7111
val Loss: 1.6737 Acc: 0.6681

Epoch 56/99
----------
train Loss: 1.2445 Acc: 0.6821
val Loss: 1.4440 Acc: 0.6853
```

```
Epoch 57/99
----------
train Loss: 1.2284 Acc: 0.6767
val Loss: 1.7736 Acc: 0.6853

Epoch 58/99
----------
train Loss: 1.1644 Acc: 0.6917
val Loss: 1.6170 Acc: 0.6724

Epoch 59/99
----------
train Loss: 1.1838 Acc: 0.6831
val Loss: 1.5307 Acc: 0.6767

Epoch 60/99
----------
train Loss: 0.9946 Acc: 0.7240
val Loss: 1.7573 Acc: 0.6853

Epoch 61/99
----------
train Loss: 1.1796 Acc: 0.7035
val Loss: 1.8040 Acc: 0.6810

Epoch 62/99
----------
train Loss: 1.1434 Acc: 0.6907
val Loss: 1.7489 Acc: 0.6940

Epoch 63/99
----------
train Loss: 1.1480 Acc: 0.6950
val Loss: 1.8842 Acc: 0.6552

Epoch 64/99
----------
train Loss: 1.0199 Acc: 0.7003
val Loss: 1.8214 Acc: 0.6767

Epoch 65/99
----------
train Loss: 1.1098 Acc: 0.7068
val Loss: 1.5391 Acc: 0.7112

Epoch 66/99
----------
train Loss: 1.2428 Acc: 0.6745
val Loss: 1.4111 Acc: 0.7198

Epoch 67/99
----------
train Loss: 1.1008 Acc: 0.7315
val Loss: 1.4569 Acc: 0.7112

Epoch 68/99
----------
```

```
train Loss: 1.0709 Acc: 0.7089
val Loss: 1.7264 Acc: 0.6724

Epoch 69/99
----------
train Loss: 1.2130 Acc: 0.7132
val Loss: 1.9483 Acc: 0.6681

Epoch 70/99
----------
train Loss: 1.2726 Acc: 0.6842
val Loss: 1.5151 Acc: 0.6724

Epoch 71/99
----------
train Loss: 1.1411 Acc: 0.7025
val Loss: 1.3938 Acc: 0.6853

Epoch 72/99
----------
train Loss: 1.1766 Acc: 0.6982
val Loss: 1.4836 Acc: 0.6509

Epoch 73/99
----------
train Loss: 1.1032 Acc: 0.7154
val Loss: 1.6212 Acc: 0.7069

Epoch 74/99
----------
train Loss: 1.1212 Acc: 0.7100
val Loss: 1.5645 Acc: 0.7026

Epoch 75/99
----------
train Loss: 1.0571 Acc: 0.7250
val Loss: 1.4115 Acc: 0.7026

Epoch 76/99
----------
train Loss: 1.0423 Acc: 0.7132
val Loss: 1.7384 Acc: 0.6422

Epoch 77/99
----------
train Loss: 1.2927 Acc: 0.6756
val Loss: 1.9079 Acc: 0.6767

Epoch 78/99
----------
train Loss: 1.0879 Acc: 0.7207
val Loss: 1.3756 Acc: 0.6853

Epoch 79/99
----------
train Loss: 1.1366 Acc: 0.7132
val Loss: 1.6116 Acc: 0.6552
```

```
Epoch 80/99
----------
train Loss: 1.1875 Acc: 0.6767
val Loss: 1.6864 Acc: 0.6681

Epoch 81/99
----------
train Loss: 1.1577 Acc: 0.7078
val Loss: 1.5772 Acc: 0.7241

Epoch 82/99
----------
train Loss: 1.1434 Acc: 0.6917
val Loss: 1.6540 Acc: 0.7026

Epoch 83/99
----------
train Loss: 1.1307 Acc: 0.7175
val Loss: 1.5359 Acc: 0.6595

Epoch 84/99
----------
train Loss: 1.0622 Acc: 0.7347
val Loss: 1.5879 Acc: 0.6293

Epoch 85/99
----------
train Loss: 1.0379 Acc: 0.7293
val Loss: 1.5679 Acc: 0.6379

Epoch 86/99
----------
train Loss: 1.1087 Acc: 0.7003
val Loss: 1.6734 Acc: 0.6767

Epoch 87/99
----------
train Loss: 1.1208 Acc: 0.6885
val Loss: 2.0424 Acc: 0.6207

Epoch 88/99
----------
train Loss: 1.1284 Acc: 0.7154
val Loss: 1.6135 Acc: 0.6940

Epoch 89/99
----------
train Loss: 1.0746 Acc: 0.7121
val Loss: 1.4978 Acc: 0.7069

Epoch 90/99
----------
train Loss: 1.0643 Acc: 0.7100
val Loss: 1.5814 Acc: 0.7155

Epoch 91/99
```

```
----------
train Loss: 1.0594 Acc: 0.7250
val Loss: 2.3255 Acc: 0.6250

Epoch 92/99
----------
train Loss: 1.1488 Acc: 0.7068
val Loss: 2.0485 Acc: 0.6336

Epoch 93/99
----------
train Loss: 1.1137 Acc: 0.7003
val Loss: 1.8076 Acc: 0.6724

Epoch 94/99
----------
train Loss: 0.9808 Acc: 0.7240
val Loss: 1.5147 Acc: 0.6810

Epoch 95/99
----------
train Loss: 1.2062 Acc: 0.7014
val Loss: 1.8300 Acc: 0.6595

Epoch 96/99
----------
train Loss: 1.1997 Acc: 0.7164
val Loss: 1.7570 Acc: 0.6767

Epoch 97/99
----------
train Loss: 1.0794 Acc: 0.7143
val Loss: 1.7037 Acc: 0.6810

Epoch 98/99
----------
train Loss: 1.0274 Acc: 0.7325
val Loss: 1.6126 Acc: 0.6724

Epoch 99/99
----------
train Loss: 1.2102 Acc: 0.6950
val Loss: 1.6786 Acc: 0.6853

Training complete in 6m 39s
Best val Acc: 0.741379
```

Question 5: Report result

1. For transfer learning used both AlexNet and Resnet50
2. Final validation accuracy achieved is 91%
3. 100 epochs are use
4. Feature extraction has made the traning faster

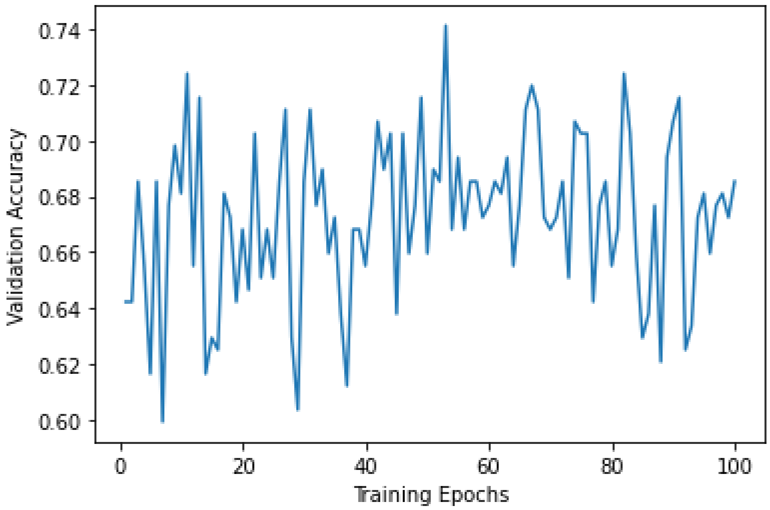
In [ ]:

In [142]:

```
best_hist = []
best_hist = [h.cpu().numpy() for h in hist]

plt.xlabel("Training Epochs")
plt.ylabel("Validation Accuracy")
plt.plot(range(1,num_epochs+1),best_hist,label="Pretrained")
```

Out[142]:

```
[<matplotlib.lines.Line2D at 0x7fc203befb38>]
```



In [143]:
```
# Initialize the non-pretrained version of the model used for this run
scratch_model,_ = initialize_model(model_name, num_classes, feature_extract=Fa
lse, use_pretrained=False)
scratch_model = scratch_model.to(device)
scratch_optimizer = optim.SGD(scratch_model.parameters(), lr=0.001, momentum=
0.9)
scratch_criterion = nn.CrossEntropyLoss()
_,scratch_hist = train_model(scratch_model, dataloaders_dict, scratch_criterio
n, scratch_optimizer, num_epochs=num_epochs)

# Plot the training curves of validation accuracy vs. number
#  of training epochs for the transfer learning method and
#  the model trained from scratch
ohist = []
shist = []

ohist = [h.cpu().numpy() for h in hist]
shist = [h.cpu().numpy() for h in scratch_hist]

plt.title("Validation Accuracy vs. Number of Training Epochs")
plt.xlabel("Training Epochs")
plt.ylabel("Validation Accuracy")
plt.plot(range(1,num_epochs+1),ohist,label="Pretrained")
plt.plot(range(1,num_epochs+1),shist,label="Scratch")
plt.ylim((0,1.))
plt.xticks(np.arange(1, num_epochs+1, 1.0))
plt.legend()
plt.show()
```

```
Epoch 0/99
----------
train Loss: 2.1974 Acc: 0.1020
val Loss: 2.1964 Acc: 0.1207

Epoch 1/99
----------
train Loss: 2.1965 Acc: 0.1214
val Loss: 2.1958 Acc: 0.1207

Epoch 2/99
----------
train Loss: 2.1963 Acc: 0.1214
val Loss: 2.1953 Acc: 0.1207

Epoch 3/99
----------
train Loss: 2.1962 Acc: 0.1203
val Loss: 2.1950 Acc: 0.1207

Epoch 4/99
----------
train Loss: 2.1958 Acc: 0.1214
val Loss: 2.1948 Acc: 0.1207

Epoch 5/99
----------
train Loss: 2.1961 Acc: 0.1214
val Loss: 2.1947 Acc: 0.1207

Epoch 6/99
----------
train Loss: 2.1953 Acc: 0.1214
val Loss: 2.1945 Acc: 0.1207

Epoch 7/99
----------
train Loss: 2.1953 Acc: 0.1214
val Loss: 2.1943 Acc: 0.1207

Epoch 8/99
----------
train Loss: 2.1954 Acc: 0.1214
val Loss: 2.1940 Acc: 0.1207

Epoch 9/99
----------
train Loss: 2.1946 Acc: 0.1214
val Loss: 2.1937 Acc: 0.1207

Epoch 10/99
----------
train Loss: 2.1947 Acc: 0.1214
val Loss: 2.1933 Acc: 0.1207

Epoch 11/99
----------
```

```
train Loss: 2.1945 Acc: 0.1214
val Loss: 2.1929 Acc: 0.1207

Epoch 12/99
----------
train Loss: 2.1942 Acc: 0.1203
val Loss: 2.1924 Acc: 0.1207

Epoch 13/99
----------
train Loss: 2.1926 Acc: 0.1160
val Loss: 2.1913 Acc: 0.1207

Epoch 14/99
----------
train Loss: 2.1918 Acc: 0.1278
val Loss: 2.1894 Acc: 0.1509

Epoch 15/99
----------
train Loss: 2.1887 Acc: 0.1300
val Loss: 2.1858 Acc: 0.1379

Epoch 16/99
----------
train Loss: 2.1815 Acc: 0.1643
val Loss: 2.1792 Acc: 0.1250

Epoch 17/99
----------
train Loss: 2.1718 Acc: 0.1450
val Loss: 2.1638 Acc: 0.1509

Epoch 18/99
----------
train Loss: 2.1634 Acc: 0.1643
val Loss: 2.1350 Acc: 0.1897

Epoch 19/99
----------
train Loss: 2.1432 Acc: 0.1869
val Loss: 2.1023 Acc: 0.2155

Epoch 20/99
----------
train Loss: 2.1257 Acc: 0.1794
val Loss: 2.0688 Acc: 0.2026

Epoch 21/99
----------
train Loss: 2.1115 Acc: 0.1933
val Loss: 2.0371 Acc: 0.2974

Epoch 22/99
----------
train Loss: 2.0752 Acc: 0.2019
val Loss: 2.0010 Acc: 0.3276
```

```
Epoch 23/99
----------
train Loss: 2.0547 Acc: 0.2331
val Loss: 1.9304 Acc: 0.2629

Epoch 24/99
----------
train Loss: 1.9912 Acc: 0.2406
val Loss: 1.7637 Acc: 0.3190

Epoch 25/99
----------
train Loss: 1.9321 Acc: 0.2685
val Loss: 1.8050 Acc: 0.3233

Epoch 26/99
----------
train Loss: 1.9024 Acc: 0.2803
val Loss: 1.6489 Acc: 0.3405

Epoch 27/99
----------
train Loss: 1.8021 Acc: 0.3136
val Loss: 1.5741 Acc: 0.3966

Epoch 28/99
----------
train Loss: 1.7593 Acc: 0.3416
val Loss: 1.4595 Acc: 0.4353

Epoch 29/99
----------
train Loss: 1.7134 Acc: 0.3631
val Loss: 1.5469 Acc: 0.5560

Epoch 30/99
----------
train Loss: 1.6103 Acc: 0.3802
val Loss: 1.4006 Acc: 0.4741

Epoch 31/99
----------
train Loss: 1.5432 Acc: 0.4082
val Loss: 1.3371 Acc: 0.4569

Epoch 32/99
----------
train Loss: 1.5699 Acc: 0.4157
val Loss: 1.3852 Acc: 0.4698

Epoch 33/99
----------
train Loss: 1.5069 Acc: 0.4028
val Loss: 1.2245 Acc: 0.5431

Epoch 34/99
```

```
----------
train Loss: 1.5013 Acc: 0.3985
val Loss: 1.3239 Acc: 0.5172

Epoch 35/99
----------
train Loss: 1.3861 Acc: 0.4662
val Loss: 1.2757 Acc: 0.5991

Epoch 36/99
----------
train Loss: 1.3703 Acc: 0.4876
val Loss: 1.0989 Acc: 0.5776

Epoch 37/99
----------
train Loss: 1.4025 Acc: 0.4382
val Loss: 1.2265 Acc: 0.5129

Epoch 38/99
----------
train Loss: 1.4081 Acc: 0.4511
val Loss: 1.1463 Acc: 0.6078

Epoch 39/99
----------
train Loss: 1.3383 Acc: 0.4694
val Loss: 1.0688 Acc: 0.6034

Epoch 40/99
----------
train Loss: 1.3010 Acc: 0.4866
val Loss: 1.2002 Acc: 0.5345

Epoch 41/99
----------
train Loss: 1.3284 Acc: 0.4973
val Loss: 1.0436 Acc: 0.6121

Epoch 42/99
----------
train Loss: 1.2835 Acc: 0.4844
val Loss: 1.0385 Acc: 0.6336

Epoch 43/99
----------
train Loss: 1.2502 Acc: 0.5274
val Loss: 1.0397 Acc: 0.6078

Epoch 44/99
----------
train Loss: 1.2372 Acc: 0.5252
val Loss: 0.9140 Acc: 0.6293

Epoch 45/99
----------
train Loss: 1.2733 Acc: 0.5027
```

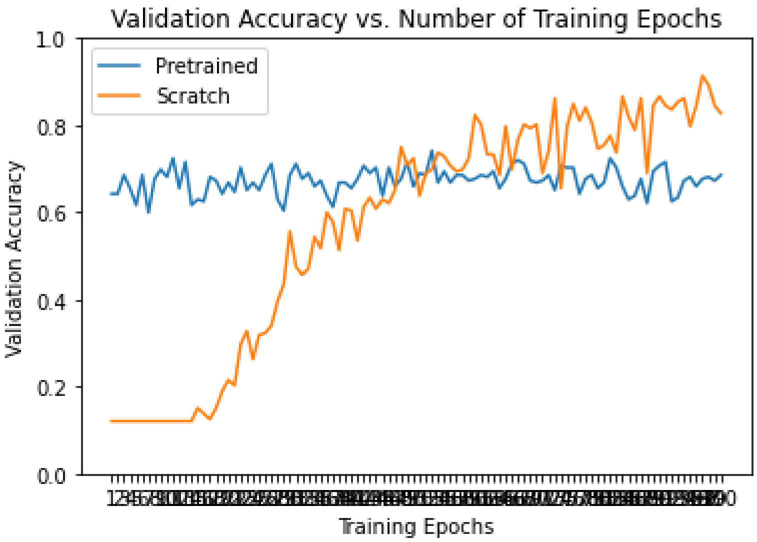
```
val Loss: 0.9769 Acc: 0.6207

Epoch 46/99
----------
train Loss: 1.2060 Acc: 0.5360
val Loss: 0.9185 Acc: 0.6509

Epoch 47/99
----------
train Loss: 1.2007 Acc: 0.5371
val Loss: 0.8735 Acc: 0.7500

Epoch 48/99
----------
train Loss: 1.1910 Acc: 0.5456
val Loss: 0.8866 Acc: 0.7069

Epoch 49/99
----------
train Loss: 1.2357 Acc: 0.5124
val Loss: 0.8796 Acc: 0.7241

Epoch 50/99
----------
train Loss: 1.1223 Acc: 0.5596
val Loss: 0.9420 Acc: 0.6379

Epoch 51/99
----------
train Loss: 1.1559 Acc: 0.5639
val Loss: 0.9121 Acc: 0.6897

Epoch 52/99
----------
train Loss: 1.1147 Acc: 0.5768
val Loss: 0.9084 Acc: 0.6983

Epoch 53/99
----------
train Loss: 1.0554 Acc: 0.5918
val Loss: 0.7749 Acc: 0.7371

Epoch 54/99
----------
train Loss: 1.0758 Acc: 0.5800
val Loss: 0.7280 Acc: 0.7284

Epoch 55/99
----------
train Loss: 1.0467 Acc: 0.5951
val Loss: 0.8639 Acc: 0.7069

Epoch 56/99
----------
train Loss: 0.9987 Acc: 0.6058
val Loss: 0.8025 Acc: 0.6940
```

```
Epoch 57/99
----------
train Loss: 1.1180 Acc: 0.5542
val Loss: 0.8720 Acc: 0.6983

Epoch 58/99
----------
train Loss: 1.0497 Acc: 0.5843
val Loss: 0.7539 Acc: 0.7241

Epoch 59/99
----------
train Loss: 1.0004 Acc: 0.6079
val Loss: 0.5998 Acc: 0.8233

Epoch 60/99
----------
train Loss: 1.0165 Acc: 0.6230
val Loss: 0.7131 Acc: 0.8017

Epoch 61/99
----------
train Loss: 0.9538 Acc: 0.6348
val Loss: 0.7580 Acc: 0.7328

Epoch 62/99
----------
train Loss: 0.9782 Acc: 0.6434
val Loss: 0.6577 Acc: 0.7328

Epoch 63/99
----------
train Loss: 0.9175 Acc: 0.6488
val Loss: 0.9726 Acc: 0.6853

Epoch 64/99
----------
train Loss: 0.8965 Acc: 0.6509
val Loss: 0.5997 Acc: 0.7974

Epoch 65/99
----------
train Loss: 0.9835 Acc: 0.6327
val Loss: 0.8052 Acc: 0.6983

Epoch 66/99
----------
train Loss: 0.9410 Acc: 0.6316
val Loss: 0.6428 Acc: 0.7672

Epoch 67/99
----------
train Loss: 0.9255 Acc: 0.6369
val Loss: 0.5768 Acc: 0.8017

Epoch 68/99
----------
```

```
train Loss: 0.8979 Acc: 0.6745
val Loss: 0.6134 Acc: 0.7931

Epoch 69/99
----------
train Loss: 0.9176 Acc: 0.6423
val Loss: 0.5810 Acc: 0.8017

Epoch 70/99
----------
train Loss: 0.8847 Acc: 0.6660
val Loss: 0.8131 Acc: 0.6897

Epoch 71/99
----------
train Loss: 0.8391 Acc: 0.6853
val Loss: 0.6666 Acc: 0.7414

Epoch 72/99
----------
train Loss: 0.8483 Acc: 0.6767
val Loss: 0.4853 Acc: 0.8621

Epoch 73/99
----------
train Loss: 0.8014 Acc: 0.7003
val Loss: 0.8836 Acc: 0.6552

Epoch 74/99
----------
train Loss: 0.8949 Acc: 0.6670
val Loss: 0.5785 Acc: 0.7974

Epoch 75/99
----------
train Loss: 0.8355 Acc: 0.6842
val Loss: 0.4474 Acc: 0.8491

Epoch 76/99
----------
train Loss: 0.7790 Acc: 0.7207
val Loss: 0.5813 Acc: 0.8103

Epoch 77/99
----------
train Loss: 0.7417 Acc: 0.7229
val Loss: 0.4964 Acc: 0.8405

Epoch 78/99
----------
train Loss: 0.8531 Acc: 0.7078
val Loss: 0.6597 Acc: 0.8060

Epoch 79/99
----------
train Loss: 0.7871 Acc: 0.7186
val Loss: 0.6388 Acc: 0.7457
```

```
Epoch 80/99
----------
train Loss: 0.7418 Acc: 0.7293
val Loss: 0.6003 Acc: 0.7543

Epoch 81/99
----------
train Loss: 0.7967 Acc: 0.7003
val Loss: 0.7403 Acc: 0.7759

Epoch 82/99
----------
train Loss: 0.7292 Acc: 0.7175
val Loss: 0.6196 Acc: 0.7371

Epoch 83/99
----------
train Loss: 0.6932 Acc: 0.7433
val Loss: 0.4582 Acc: 0.8664

Epoch 84/99
----------
train Loss: 0.7259 Acc: 0.7315
val Loss: 0.5178 Acc: 0.8190

Epoch 85/99
----------
train Loss: 0.6729 Acc: 0.7422
val Loss: 0.5911 Acc: 0.7888

Epoch 86/99
----------
train Loss: 0.6582 Acc: 0.7497
val Loss: 0.4590 Acc: 0.8621

Epoch 87/99
----------
train Loss: 0.6360 Acc: 0.7626
val Loss: 1.0279 Acc: 0.6897

Epoch 88/99
----------
train Loss: 0.6899 Acc: 0.7293
val Loss: 0.5316 Acc: 0.8448

Epoch 89/99
----------
train Loss: 0.6584 Acc: 0.7508
val Loss: 0.3800 Acc: 0.8664

Epoch 90/99
----------
train Loss: 0.6964 Acc: 0.7433
val Loss: 0.4764 Acc: 0.8448

Epoch 91/99
```

```
----------
train Loss: 0.6022 Acc: 0.7787
val Loss: 0.5325 Acc: 0.8362

Epoch 92/99
----------
train Loss: 0.6199 Acc: 0.7744
val Loss: 0.4761 Acc: 0.8534

Epoch 93/99
----------
train Loss: 0.6266 Acc: 0.7830
val Loss: 0.4236 Acc: 0.8621

Epoch 94/99
----------
train Loss: 0.7157 Acc: 0.7487
val Loss: 0.7216 Acc: 0.7974

Epoch 95/99
----------
train Loss: 0.6262 Acc: 0.7658
val Loss: 0.3939 Acc: 0.8448

Epoch 96/99
----------
train Loss: 0.6879 Acc: 0.7411
val Loss: 0.3397 Acc: 0.9138

Epoch 97/99
----------
train Loss: 0.6653 Acc: 0.7562
val Loss: 0.3737 Acc: 0.8922

Epoch 98/99
----------
train Loss: 0.6175 Acc: 0.7669
val Loss: 0.4746 Acc: 0.8448

Epoch 99/99
----------
train Loss: 0.6143 Acc: 0.7583
val Loss: 0.5713 Acc: 0.8276

Training complete in 8m 3s
Best val Acc: 0.913793
```


Validation Accuracy vs. Number of Training Epochs
Pretrained
Scratch
Validation Accuracy
Training Epochs
1.0
0.8
0.6
0.4
0.2
0.0