CAPTCHA Bypass

Team 99

Nicholas Leung

Coden Mercurius

Pranavbhai Patel

Ravi Singh

Description

CAPTCHA, or Completely Automated Public Turing Test to Tell Computers and Humans Apart, is a challenge-response test that determines whether a user is authentic (human) or inauthentic (machine). They require users to authenticate themselves by retyping a character sequence prior to completing a request. This notebook implements a CAPTCHA bypass using deep learning. The team aims to investigate weaknesses and vulnerabilities of the CAPTCHA system.

In [185]:
```
%%shell
jupyter nbconvert --to html /content/captcha.ipynb
```

```
[NbConvertApp] Converting notebook /content/captcha.ipynb to html
[NbConvertApp] Writing 695383 bytes to /content/captcha.html
```

Out[185]:

In [2]:
```
# Imports
import torch
import torch.nn as nn
import os
from skimage import io
from torch.utils.data import Dataset, DataLoader
import torch.utils.data
import torchvision
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
import time
```

Part 1. Data Processing

The dataset for this model is generated using the following library: [https://github.com/lepture/captcha (https://github.com/lepture/captcha)](https://github.com/lepture/captcha) and automated by the script `dataset_generator.py` .

The character space began as purely numeric (0-9) but has since expanded to become alphanumeric (0-9, A-Z). Alphabetical characters are capitalized. Characters are uniformly distributed in terms of occurrence in the dataset.

The generated dataset `alphanumeric_dataset.zip` is availiable on the private team Google Drive because it is too large for the Github repository. Upload `alphanumeric_dataset.zip` into the Colab Files and unzip.

In [3]:
```
# Unzip dataset
!unzip  -qq /content/alphanumeric_dataset.zip -d /content/
```

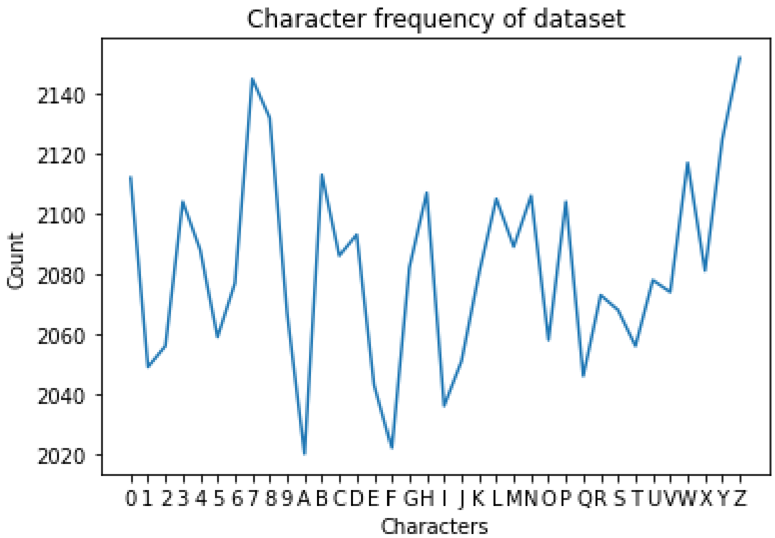
In [5]:
```
class CaptchaDataset(Dataset):
  """ Captcha Dataset """

  def __init__(self, directory):
    self.directory = directory
    self.captchas = os.listdir(directory)
    self.captchas.remove("metadata.txt")

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

    self.character_set = open(directory + "/metadata.txt", "r").readline().spl
it(',')
    self.characters_to_identifier = {}

    for i in range(len(self.character_set)):
      self.characters_to_identifier.update({ self.character_set[i]: i })

  def __len__(self):
    # Assumes each file in the dataset directory represents a data sample
    return len(self.captchas)

  def __getitem__(self, index):
    sample_name = self.captchas[index]
    sample_captcha_values = list(sample_name[0:-4]) # Slice s.t. remove png fi
le extension

    # Read the image and represent it as a tensor
    image = io.imread(self.directory + '/' + sample_name)
    image = self.transform(image)

    # Represent each character as an integer identifier
    label = []
    for char in sample_captcha_values:
      label.append(self.characters_to_identifier.get(char))

    return (image, torch.tensor(label))
```

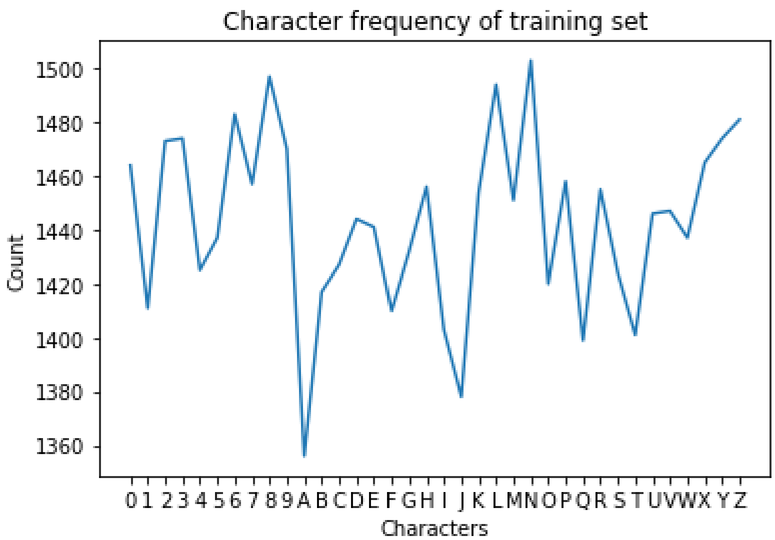
In [6]:
```
dataset_path = "/content/alphanumeric_dataset"

# Instantiate dataset
dataset = CaptchaDataset(dataset_path)
```

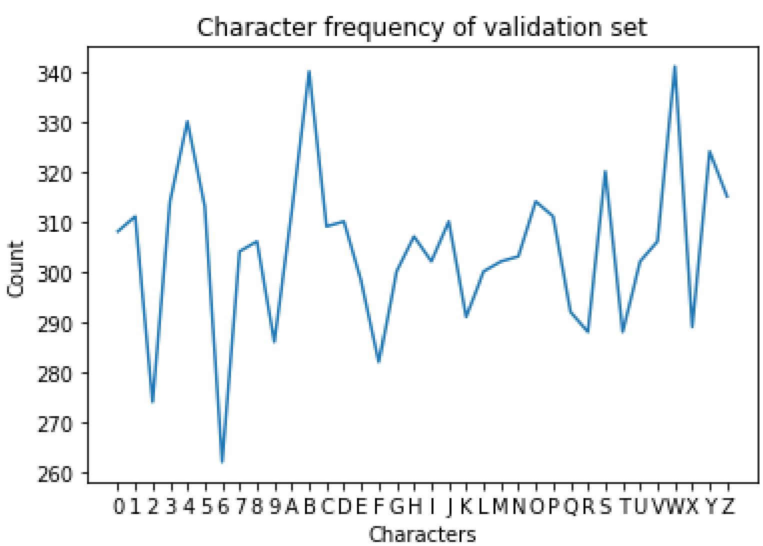
In [7]:
```
def visualize_character_frequency(dataloader, title):
  character_frequency = {} # Contains frequency information
  character_set = dataset.character_set

  # Populate character_frequency
  for _, labels in dataloader:
    for label in labels:
      for char_identifier in label:
        char = character_set[char_identifier.item()]
        current_value = character_frequency.get(char, None)

        if current_value is None:
          character_frequency.update({ char : 0 })
        else:
          character_frequency.update({ char : current_value + 1 })

  x_values = range(len(character_set))
  y_values = []

  for char in character_set:
    count = character_frequency.get(char)
    y_values.append(count)

  plt.title(title)
  plt.plot(x_values, y_values)
  plt.xlabel("Characters")
  plt.ylabel("Count")
  plt.xticks(x_values, character_set)
  plt.show()
```

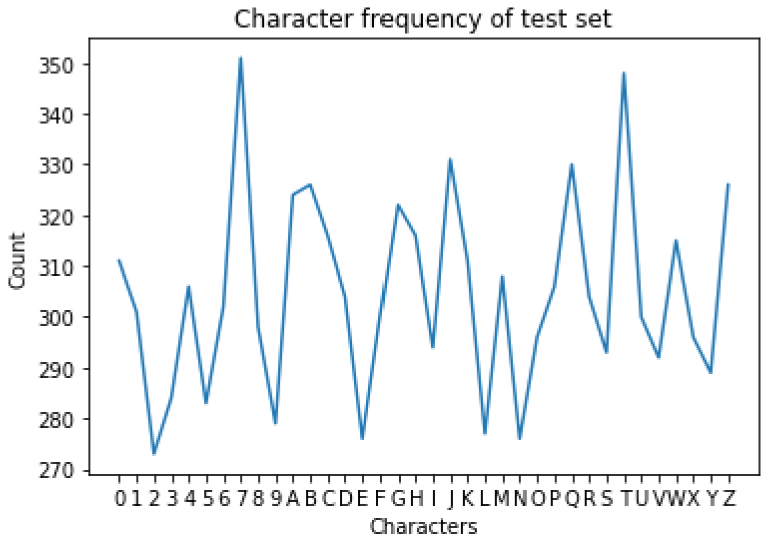
In [8]:
```
def get_data_loaders(dataset, batch_size, total_size=None):

  if total_size is None:
    total_size = len(dataset)

  training_ratio = 0.7
  validation_ratio = 0.15
  # test_ratio implied

  train_length = int(total_size * training_ratio)
  validation_length = int((total_size - train_length) * (validation_ratio / (
1 - training_ratio )))
  test_length = total_size - train_length - validation_length
  fill = len(dataset) - total_size

  train_set, valid_set, test_set, fill_set = torch.utils.data.random_split(dat
aset, [train_length, validation_length, test_length, fill], torch.Generator().
manual_seed(10))

  train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size,
num_workers=1, drop_last=True, shuffle=True)
  valid_loader = torch.utils.data.DataLoader(valid_set, batch_size=batch_size,
num_workers=1, drop_last=True, shuffle=True)
  test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, n
um_workers=1, drop_last=True, shuffle=True)

  return train_loader, valid_loader, test_loader
```

In [9]:
```
# Dataset visualization
train, valid, test = get_data_loaders(dataset, 100)

visualize_character_frequency(torch.utils.data.DataLoader(dataset, num_workers
=1), title="Character frequency of dataset")
visualize_character_frequency(train, title="Character frequency of training se
t")
visualize_character_frequency(valid, title="Character frequency of validation
 set")
visualize_character_frequency(test, title="Character frequency of test set")
```


Character frequency of dataset
Count
2020 2040 2060 2080 2100 2120 2140
0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Characters


Character frequency of training set
Count
1360 1380 1400 1420 1440 1460 1480 1500
0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Characters


Character frequency of validation set
Count
260 270 280 290 300 310 320 330 340
0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Characters



Part 2. Character Segmentation

Character segmentation must occur prior to character classification. This entails using `OpenCV.findContours()` to perform blob detection on a CAPTCHA image input. This will extract each individual character (5 total) for input to the model.

If characters are overlapping, the median width is assumed to separate the characters.

In [10]:
```
import random
import cv2
import torchvision as tv
```

In [11]:
```
def processimage(image, thresh):
      #Format image type/ dimensions
      image=image.permute(1,2,0)
      image=image.numpy()
      imageorig=image

      #Modify image so contours/ borders can be easily found
      #Greyscale
      image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
      #Binary Threshold
      NA, image = cv2.threshold(image, thresh, 1, cv2.THRESH_BINARY)
      #Erosion
      kernel = np.ones((2,2),np.uint8)
      image = cv2.dilate(image,kernel,iterations = 1)
      #Vertical Blur and Resharpen
      morpher = cv2.getStructuringElement(cv2.MORPH_RECT, (1,3))
      image = cv2.morphologyEx(image, cv2.MORPH_CLOSE, morpher)
      #Binary Threshold
      thresh, image = cv2.threshold(image,thresh, 1, cv2.THRESH_BINARY)
      #Expand Border
      image=cv2.copyMakeBorder(image, 5, 5, 5, 5,cv2.BORDER_CONSTANT,value=1)
      image = image.astype(np.uint8)

      return imageorig,image
```

In [12]:
```
def segmentimage(image,narrow):
    #Return list of borderlines in image
    contours, hierarchy = cv2.findContours(image, cv2.RETR_TREE, cv2.CHAIN_APP
ROX_SIMPLE)
    #imagecont=cv2.drawContours(image, contours, -1, (0, 0.5, 0), 1)

    #Creates boxes for every large object
    boxes=[]
    for contour in contours:
        [x,y,w,h]=cv2.boundingRect(contour)
        if(w>8 and w<120 and h>30):
          boxes.append([x,y,w,h])
    boxes.sort(key=lambda x: x[0])

    #Eliminates boxes that are contained whithin other boxes (subparts of a le
tter)
    i=1
    while (i<len(boxes)):
      box=boxes[i]
      boxprev=boxes[i-1]
      if (box[0]>boxprev[0] and box[1]>boxprev[1] and (box[0]+box[2])<(boxprev
[0]+boxprev[2]) and (box[1]+box[3])<(boxprev[1]+boxprev[3])):
        boxes.pop(i)
        i-=1
      i+=1

    #If boxes are too wide they may contain multiple boxes
    #They are split vertically into 2 or 3 subboxes (even width splits)
    i=0
    while (i<len(boxes)):
      box=boxes[i]
      if (box[2]>box[3]*(0.95-narrow)):
        x,y,w,h=boxes.pop(i)
        boxes.insert(i,[x+int((2*w)/3),y,int(w/3),h])
        boxes.insert(i,[x+int((w)/3),y,int(w/3),h])
        boxes.insert(i,[x,y,int(w/3),h])
      elif (box[2]>box[3]*(0.6-narrow)):
        x,y,w,h=boxes.pop(i)
        boxes.insert(i,[x+int(w/2),y,int(w/2),h])
        boxes.insert(i,[x,y,int(w/2),h])
      if (i>=len(boxes)-1):
        break
      i+=1

    return boxes
```

In [14]:
```
def getcharacterimages(images, showsegments=False, filterBadSegmentation=False
, imgsize=80):
  characters=[]
  for i in range (0,len(images)):
    imageraw = images[i]

    # CAPTCHA image pre-processed, custom function called
    # Rectangle Borders of each character obtained, custom function called
    for i in range(0,7):
      thresh, narrow = 0.6,0
      if (i>=1):
        narrow=0.1
        thresh=0.6+(i-1)*0.1
      imageorig, image=processimage(imageraw,thresh)
      imageboxes = np.copy(image)
      boxes=segmentimage(image,narrow)
      if (len(boxes)>=5):
        break

    # Filter bad segmentation cases
    if filterBadSegmentation and len(boxes) < 5:
      continue

    # Individual Characters images are cut out from CAPTCHA image
    charactersset=[]
    for i in range(0,5):
      # If insufficient letters obtainable, add an empty image
      if (i<len(boxes)):
        box=boxes[i]
      else:
        box=[0,0,1,1]

      [x,y,w,h]=box
      char=image[y:y+h,x:x+w]
      height=char.shape[0]
      width=char.shape[1]
      # cv2.copyMakeBorder(soruce, top, bottom, left, right, borderType, valu
e)
      if (width>height):
        char=cv2.copyMakeBorder(char, int ((width-height)/2),int ((width-heigh
t)/2), 0, 0,cv2.BORDER_CONSTANT,value=1)
      if (height>width):
        char=cv2.copyMakeBorder(char,0,0, int ((height-width)/2), int ((height
-width)/2),cv2.BORDER_CONSTANT,value=1)
      char = cv2.resize(char, dsize=(imgsize, imgsize), interpolation=cv2.INTE
R_CUBIC)
      char=torch.Tensor(char)
      charactersset.append(char)
      cv2.rectangle(imageboxes,(x,y),(x+w,y+h),0,1)
    charactersset=torch.stack(charactersset)
    characters.append(charactersset)

    if (showsegments==True):
      plt.imshow(imageorig)
      plt.show()
      plt.imshow(imageboxes, cmap='gray', vmin = 0, vmax = 1)
```

```
      plt.show()
      for i in range(0,5):
        plt.subplot(1,5,i+1)
        plt.imshow(charactersset[i], cmap='gray', vmin = 0, vmax = 1)
      plt.show()

  return torch.stack(characters)
```

In [15]:
```
"""
How To Use - getcharacterimages(images, showsegments=False)

Input = tensor(batchsize,numchannels,height,width) (see below)
Output = tensor(batchsize, numcharacters = 5, height = 80, width = 80 )

Set `showsegments` to `True`to visualize segmentation
"""
train, valid, test = get_data_loaders(dataset, 100)

images_processed = 0
successful_segmentation = 0

for images, labels in train:
  characters = getcharacterimages(images, False, True)
  images_processed = images_processed + len(labels)
  successful_segmentation = successful_segmentation + len(characters)

print(f"Successful segmentation accuracy: { successful_segmentation / images_p
rocessed }")
```

```
Successful segmentation accuracy: 0.9701923076923077
```

Part 3. Base Model

The base model is a non-deep learning method. The base model leverages the same character segmentation module and an SVM architecture is used for character classification.

The base model is to serve as a baseline of comparison for the primary model.

In [16]:
```
from sklearn import svm
import numpy as np
```

In [17]:
```
class BaseModel:

  def __init__(self):
    self.classifier = svm.SVC()

  def fit_classifier(self, dataloader):

    # Preprocessing to make our PyTorch data in acceptable format

    input_acc = []
    labels_acc = []

    for images, labels in dataloader:

      segmented_captchas = getcharacterimages(images, imgsize=28)

      # Iterate over each captcha
      for i in range(len(segmented_captchas)):
        captcha = segmented_captchas[i]

        # Iterate over each character
        for j in range(len(captcha)):
          input_acc.append(captcha[j].detach().numpy().reshape(-1))
          labels_acc.append(labels[i][j].detach().numpy())

    input_acc = np.array(input_acc)
    labels_acc = np.array(labels_acc)

    # Train character classification

    self.classifier.fit(input_acc, labels_acc)

  def predict(self, images):
    segmented_captchas = getcharacterimages(images, imgsize=28)

    output = []
    for captcha in segmented_captchas:

      out_captcha = []

      for character in captcha:
        numpy_char = character.detach().numpy().reshape((1, -1)) # Reshape to acceptable input for SVM predict()
        out_char = self.classifier.predict(numpy_char)
        out_captcha.append(out_char.item())

      output.append(out_captcha)

    return torch.tensor(output)
```

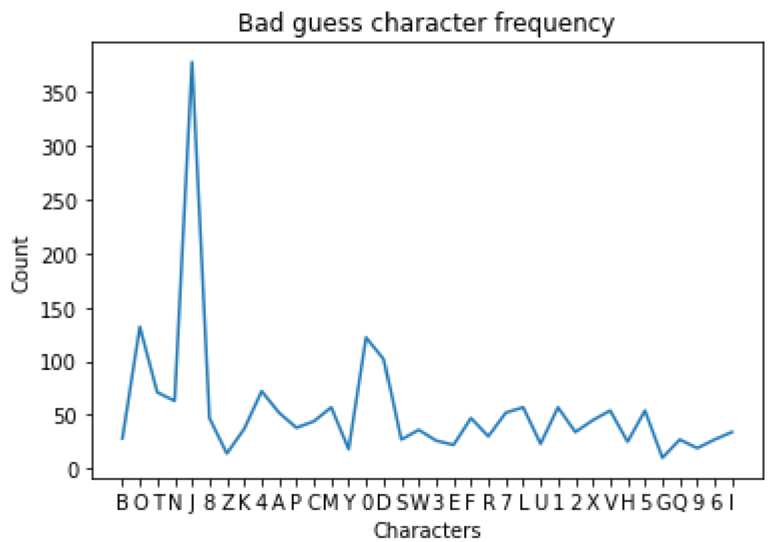
In [18]:
```
model = BaseModel()
train_small, valid_small, test_small = get_data_loaders(dataset, 100, 3000)

model.fit_classifier(train_small)
```

In [21]:
```
def evaluate_base_model(dataloader):

  total_character_guesses = 0
  total_captcha_guesses = 0

  incorrect_character_guesses = 0
  incorrect_captcha_guesses = 0

  failed_guess_frequency = {}

  for images, labels in dataloader:
    out = model.predict(images)

    # Iterate through each sample captcha in batch
    for i in range(len(labels)):
      bad_guess = False

      # Iterate through each character of captcha
      for j in range(len(labels[i])):

        total_character_guesses = total_character_guesses + 1
        guess = out[i][j]
        expected = labels[i][j]

        if (guess != expected):
          incorrect_character_guesses = incorrect_character_guesses + 1

          # Track per character bad guesses
          current_failed_guess_count = failed_guess_frequency.get(dataset.char
acter_set[guess], 0)
          failed_guess_frequency.update({ dataset.character_set[guess]: curren
t_failed_guess_count + 1 })

          bad_guess = True

      if bad_guess:
        incorrect_captcha_guesses = incorrect_captcha_guesses + 1

      total_captcha_guesses = total_captcha_guesses + 1

  # Overall accuracy information

  character_guess_accuracy = (total_character_guesses - incorrect_character_gu
esses) / total_character_guesses
  captcha_guess_accuracy = (total_captcha_guesses - incorrect_captcha_guesses)
/ total_captcha_guesses

  print(f"Character Accuracy: {character_guess_accuracy}")
  print(f"Captcha Accuracy: {captcha_guess_accuracy}")

  # Plot incorrect character guess frequency

  bad_guess_character_set = failed_guess_frequency.keys()

  x_values = range(len(bad_guess_character_set))
  y_values = []
```

```
    for char in bad_guess_character_set:
      count = failed_guess_frequency.get(char)
      y_values.append(count)

    plt.title("Bad guess character frequency")
    plt.plot(x_values, y_values)
    plt.xlabel("Characters")
    plt.ylabel("Count")
    plt.xticks(x_values, bad_guess_character_set)
    plt.show()
```

In [22]:
```
evaluate_base_model(valid)
```

```
Character Accuracy: 0.8199090909090909
Captcha Accuracy: 0.5095454545454545
```



Part 4: Primary Model

The primary model is a standard CNN with two convolutional layers and two fully-connected layers. Each convolutional layer is coupled with a max pooling layer (stride 2).

In [27]:
```
import torch.nn.functional as F
import torch.optim as optim
import pandas as pd
```

In [28]:
```
# Sample CNN based on Lab 2 model
class CaptchaCNN(nn.Module):
    def __init__(self):
        super(CaptchaCNN, self).__init__()
        self.name = "CaptchaCNN"

        # 5 chars, 5,5 we decide
        self.conv1 = nn.Conv2d(1, 5, 5)
        # w,h =
        self.pool1 = nn.MaxPool2d(2, 2)
        # w,h =

        # 5 matches the second 5 from the first CNN layer
        self.conv2 = nn.Conv2d(5, 10, 5)
        # w,h =
        self.pool2 = nn.MaxPool2d(2,2)
        # w,h =

        self.fc1 = nn.Linear(160, 32)
        self.fc2 = nn.Linear(32, 36)

    def forward(self, img):
        x = getcharacterimages(img, False, imgsize=28)
        x = x.reshape(-1, 1, 28, 28)
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(-1, 160)
        return self.fc2(F.relu(self.fc1(x)))
```

In [29]:
```
def plot(title, xlabel, ylabel, data1, data1_label, data2, data2_label, epochs
):
    plt.title(title)
    plt.plot(epochs, data1, label=data1_label)
    if data2 is not None:
      plt.plot(epochs, data2, label=data2_label)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.legend(loc='best')
    plt.show()
```

In [30]:
```
def get_accuracy(model, data_loader):
  total = 0
  correct = 0
  char_correct = 0
  captcha_length = 5
  for imgs, labels in data_loader:
      if torch.cuda.is_available():
          imgs = imgs.cuda()
          labels = labels.cuda()
      for i in range(batch_size):
        output = model(imgs[i].unsqueeze(dim=0))
        pred = output.max(1, keepdim=True)[1] # get the index of the max logit
        num_correct = 0
        for j in range(captcha_length):
          # print(labels[i][j])
          # print(pred[j])
          if labels[i][j] == pred[j].squeeze(0):
            num_correct += 1
            char_correct += 1
          # print(num_correct)
        if num_correct == 5:
          correct += 1
        total += 1
  return correct / total, char_correct / (5 * total)
```

In [31]:
```
def get_model_name(name, epoch, learning_rate=1e-4):
  """
  Generate a name for the model consisting of all the hyperparameter values
  """
  path="model_{0}_lr{1}_epoch{2}".format(
      name, learning_rate, epoch)
  return path
```

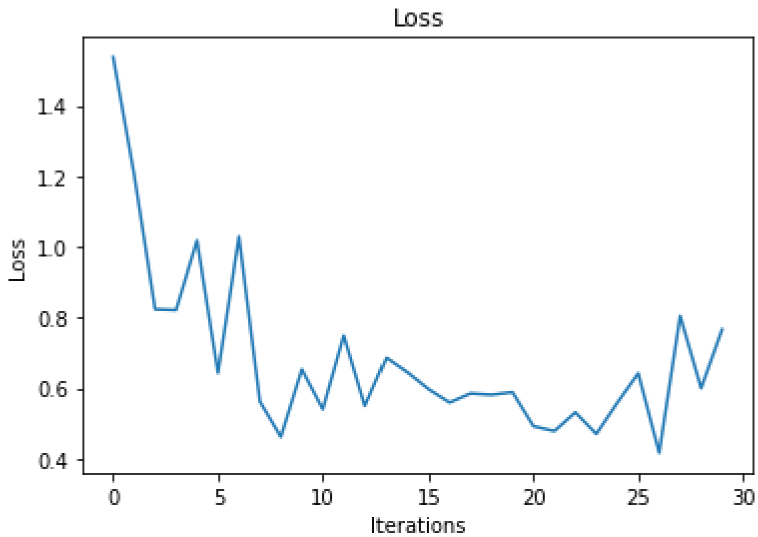
In [106]:
```
def train_cnn(model, x, y, batch_size=128, num_epochs=20, learning_rate=0.001
):
    torch.manual_seed(360)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    train_loader = x
    valid_loader = y

    iters = []
    losses = []
    train_acc = []
    valid_acc = []

    epoch = 0

    for epoch in range(num_epochs):
        for imgs, labels in train_loader:
            labels = labels.reshape(-1)
            if use_cuda and torch.cuda.is_available():
                imgs = imgs.cuda()
                labels = labels.cuda()
            out = model(imgs) # forward pass
            loss = criterion(out, labels) # compute the total loss
            loss.backward()  # backward pass (compute parameter updates)
            optimizer.step()  # make the updates for each parameter
            optimizer.zero_grad()  # a clean up step for PyTorch

        # Save the current model (checkpoint) to a file
        model_path = get_model_name(model.name, epoch, learning_rate=learning_
rate)
        torch.save(model.state_dict(), model_path)

        # save the current training information
        iters.append(epoch)
        losses.append(float(loss))  # compute *average* loss
        captcha_acc, char_acc = get_accuracy(model, train_loader)
        train_acc.append(captcha_acc)  # compute training accuracy
        if y != None:
            valid_acc.append(
                get_accuracy(model, valid_loader)[0]
            )  # compute validation accuracy
        if y != None:
            print(
                (
                    "Epoch {}: Character accuracy: {}, Training accuracy: {},
 " + "Validation accuracy: {}"
                ).format(epoch + 1, char_acc, train_acc[epoch], valid_acc[epoc
h])
            )
        else:
            print(
                ("Epoch {}: Character accuracy: {}, Training accuracy: {}").fo
rmat(epoch + 1, char_acc, train_acc[epoch])
            )
```

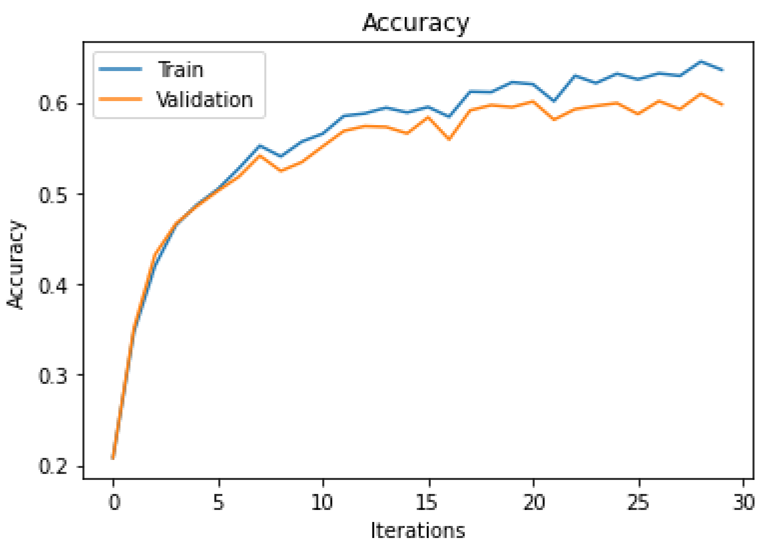
```
        epoch += 1

    plt.title("Loss")
    plt.plot(iters, losses, label="Train")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()

    plt.title("Accuracy")
    plt.plot(iters, train_acc, label="Train")
    if y != None:
        plt.plot(iters, valid_acc, label="Validation")
    plt.xlabel("Iterations")
    plt.ylabel("Accuracy")
    plt.legend(loc="best")
    plt.show()

    if y != None:
        return losses, train_acc, valid_acc
    return losses, train_acc
```

In [107]:
```
model = CaptchaCNN()
train, valid, test = get_data_loaders(dataset, 32)
losses, train_acc, valid_acc = train_cnn(model, train, valid, 32, 30, 0.001)
```

```
Epoch 1: Character accuracy: 0.685766006097561, Training accuracy: 0.20807926
829268292, Validation accuracy: 0.2080357142857143
Epoch 2: Character accuracy: 0.7602705792682927, Training accuracy: 0.3469893
292682927, Validation accuracy: 0.3513392857142857
Epoch 3: Character accuracy: 0.7902248475609757, Training accuracy: 0.4193025
9146341464, Validation accuracy: 0.4316964285714286
Epoch 4: Character accuracy: 0.8091844512195122, Training accuracy: 0.4643673
780487805, Validation accuracy: 0.4660714285714286
Epoch 5: Character accuracy: 0.8171684451219512, Training accuracy: 0.4868521
341463415, Validation accuracy: 0.48526785714285714
Epoch 6: Character accuracy: 0.8233612804878049, Training accuracy: 0.5041920
731707317, Validation accuracy: 0.5022321428571429
Epoch 7: Character accuracy: 0.8323170731707317, Training accuracy: 0.5270579
268292683, Validation accuracy: 0.5178571428571429
Epoch 8: Character accuracy: 0.8409870426829268, Training accuracy: 0.5521150
914634146, Validation accuracy: 0.5410714285714285
Epoch 9: Character accuracy: 0.8378239329268292, Training accuracy: 0.5402057
926829268, Validation accuracy: 0.5241071428571429
Epoch 10: Character accuracy: 0.8426067073170732, Training accuracy: 0.556592
9878048781, Validation accuracy: 0.5339285714285714
Epoch 11: Character accuracy: 0.8457507621951219, Training accuracy: 0.565358
231707317, Validation accuracy: 0.5513392857142857
Epoch 12: Character accuracy: 0.8515434451219512, Training accuracy: 0.584889
481707317, Validation accuracy: 0.5683035714285715
Epoch 13: Character accuracy: 0.8530678353658536, Training accuracy: 0.587652
4390243902, Validation accuracy: 0.5736607142857143
Epoch 14: Character accuracy: 0.8548208841463415, Training accuracy: 0.594035
8231707317, Validation accuracy: 0.5727678571428572
Epoch 15: Character accuracy: 0.8534679878048781, Training accuracy: 0.588700
4573170732, Validation accuracy: 0.565625
Epoch 16: Character accuracy: 0.8560403963414634, Training accuracy: 0.594798
018292683, Validation accuracy: 0.5834821428571428
Epoch 17: Character accuracy: 0.8531440548780488, Training accuracy: 0.584032
0121951219, Validation accuracy: 0.5589285714285714
Epoch 18: Character accuracy: 0.8606897865853659, Training accuracy: 0.611756
8597560976, Validation accuracy: 0.5910714285714286
Epoch 19: Character accuracy: 0.8612804878048781, Training accuracy: 0.611185
2134146342, Validation accuracy: 0.596875
Epoch 20: Character accuracy: 0.8640625, Training accuracy: 0.62214176829268
3, Validation accuracy: 0.5946428571428571
Epoch 21: Character accuracy: 0.863795731707317, Training accuracy: 0.6199504
573170732, Validation accuracy: 0.6008928571428571
Epoch 22: Character accuracy: 0.8588414634146342, Training accuracy: 0.601181
4024390244, Validation accuracy: 0.5808035714285714
Epoch 23: Character accuracy: 0.8660251524390243, Training accuracy: 0.629382
6219512195, Validation accuracy: 0.5924107142857142
Epoch 24: Character accuracy: 0.8645198170731707, Training accuracy: 0.621189
0243902439, Validation accuracy: 0.5959821428571429
Epoch 25: Character accuracy: 0.867701981707317, Training accuracy: 0.6315739
329268293, Validation accuracy: 0.5991071428571428
Epoch 26: Character accuracy: 0.8661204268292683, Training accuracy: 0.625285
8231707317, Validation accuracy: 0.5870535714285714
Epoch 27: Character accuracy: 0.8681021341463414, Training accuracy: 0.631955
0304878049, Validation accuracy: 0.6013392857142857
Epoch 28: Character accuracy: 0.8666920731707317, Training accuracy: 0.629192
0731707317, Validation accuracy: 0.5924107142857142
Epoch 29: Character accuracy: 0.8721989329268293, Training accuracy: 0.645007
```

```
6219512195, Validation accuracy: 0.609375
Epoch 30: Character accuracy: 0.8690167682926829, Training accuracy: 0.635861
2804878049, Validation accuracy: 0.5977678571428572
```





Part 5: AlexNet Transfer Learning

The pretrained AlexNet model was imported as a backup to the primary model if the CNN architecture yielded subpar results. Similar to the primary model, the AlexNet model was coupled with the character segmentation module.

In [73]:
```
import torchvision.models
alexnet = torchvision.models.alexnet(pretrained=True)

AlexNet_train, AlexNet_valid = [], []

train, valid, test = get_data_loaders(dataset, 100)

for images, labels in train:
    AlexNet_train.append((getcharacterimages(images, False, imgsize=80), label
s))

for images, labels in valid:
    AlexNet_valid.append((getcharacterimages(images, False, imgsize=80), label
s))
```

In [74]:
```
imgs_train, labels_train = [], []
imgs_valid, labels_valid = [], []

for img, label in AlexNet_train:
    img_grey = img.reshape(-1, 1, 80, 80)
    label = label.reshape(-1)
    img_color = img_grey.repeat(1,3,1,1)
    features = torch.from_numpy(alexnet.features(img_color).detach().numpy())
    imgs_train.append(features)
    labels_train.append(label)

for img, label in AlexNet_valid:
    img = img.reshape(-1, 1, 80, 80)
    label = label.reshape(-1)
    img_color = img.repeat(1,3,1,1)
    features = torch.from_numpy(alexnet.features(img_color).detach().numpy())
    imgs_valid.append(features)
    labels_valid.append(label)

AlexNet_train = list(zip(imgs_train, labels_train))
AlexNet_valid = list(zip(imgs_valid, labels_valid))
```

In [60]:
```
def get_accuracy_alexnet(model, data_loader):
  total = 0
  correct = 0
  char_correct = 0
  captcha_length = 5
  for imgs, labels in data_loader:
      if torch.cuda.is_available():
          imgs = imgs.cuda()
          labels = labels.cuda()
      output = model(imgs)
      pred = output.max(1, keepdim=True)[1].squeeze(1) # get the index of the
 max logit
      for i in range(0, len(output), captcha_length):
        num_correct = 0
        for j in range(captcha_length):
          if labels[i+j] == pred[i+j]:
            num_correct += 1
            char_correct += 1
        if num_correct == 5:
          correct += 1
        total += 1
  return correct / total, char_correct / (5 * total)

def train_alexnet(model, x, y, batch_size=128, num_epochs=20, learning_rate=0.
001):
    torch.manual_seed(360)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    train_loader = x
    valid_loader = y

    iters = []
    losses = []
    train_acc = []
    valid_acc = []

    epoch = 0

    for epoch in range(num_epochs):
        for img, label in train_loader:
            if use_cuda and torch.cuda.is_available():
                img = img.cuda()
                label = label.cuda()
            out = model(img) # forward pass
            loss = criterion(out, label) # compute the total loss
            loss.backward()  # backward pass (compute parameter updates)
            optimizer.step()  # make the updates for each parameter
            optimizer.zero_grad()  # a clean up step for PyTorch

        # Save the current model (checkpoint) to a file
        model_path = get_model_name(model.name, epoch, learning_rate=learning_
rate)
        torch.save(model.state_dict(), model_path)
```

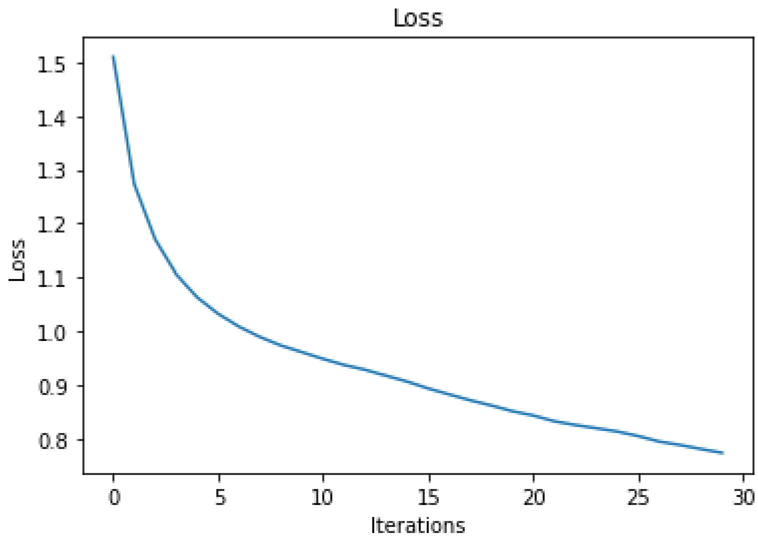
```
        # save the current training information
        iters.append(epoch)
        losses.append(float(loss))  # compute *average* loss
        captcha_acc, char_acc = get_accuracy_alexnet(model, train_loader)
        train_acc.append(captcha_acc)  # compute training accuracy
        if y != None:
            captcha_acc, char_acc = get_accuracy_alexnet(model, valid_loader)
# compute validation accuracy
            valid_acc.append(captcha_acc)
        if y != None:
            print(
                (
                    "Epoch {}: Character accuracy: {}, Training accuracy: {},
 " + "Validation accuracy: {}"
                ).format(epoch + 1, char_acc, train_acc[epoch], valid_acc[epoc
h])
            )
        else:
            print(
                ("Epoch {}: Character accuracy: {}, Training accuracy: {}").fo
rmat(epoch + 1, char_acc, train_acc[epoch])
            )
        epoch += 1

        # model_path = get_model_name(model.name, batch_size, learning_rate, e
poch)
        # torch.save(model.state_dict(), model_path)
    plt.title("Loss")
    plt.plot(iters, losses, label="Train")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()

    plt.title("Accuracy")
    plt.plot(iters, train_acc, label="Train")
    if y != None:
        plt.plot(iters, valid_acc, label="Validation")
    plt.xlabel("Iterations")
    plt.ylabel("Accuracy")
    plt.legend(loc="best")
    plt.show()

    if y != None:
        return losses, train_acc, valid_acc
    return losses, train_acc
```

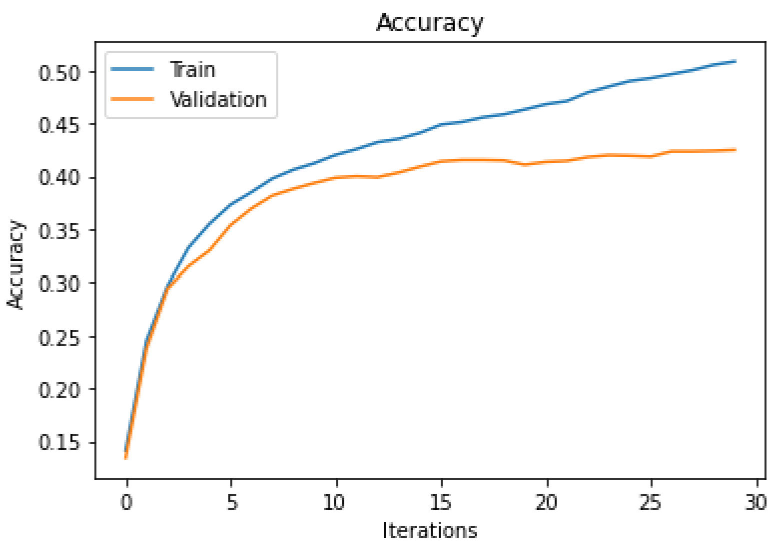
In [56]:
```
class AlexNetANNClassifier(nn.Module):
    def __init__(self):
        super(AlexNetANNClassifier, self).__init__()
        self.name = "AlexNetANNClassifier"
        self.fc1 = nn.Linear(256, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 36)

    def forward(self, img):
        x = img.view(-1, 256)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        return x
```

In [75]:
```
model = AlexNetANNClassifier()
if torch.cuda.is_available():
    model.cuda()

losses, train_acc, valid_acc = train_alexnet(model, AlexNet_train, AlexNet_valid, 32, 30, 0.001)
```

```
Epoch 1: Character accuracy: 0.6286363636363637, Training accuracy: 0.1411538
4615384616, Validation accuracy: 0.1340909090909091
Epoch 2: Character accuracy: 0.7095454545454546, Training accuracy: 0.2454807
6923076922, Validation accuracy: 0.23772727272727273
Epoch 3: Character accuracy: 0.7362727272727273, Training accuracy: 0.2968269
230769231, Validation accuracy: 0.29409090909090907
Epoch 4: Character accuracy: 0.7480909090909091, Training accuracy: 0.3325961
538461538, Validation accuracy: 0.315
Epoch 5: Character accuracy: 0.7565454545454545, Training accuracy: 0.355, Va
lidation accuracy: 0.33
Epoch 6: Character accuracy: 0.7648181818181818, Training accuracy: 0.3730769
230769231, Validation accuracy: 0.35363636363636364
Epoch 7: Character accuracy: 0.7711818181818182, Training accuracy: 0.385, Va
lidation accuracy: 0.36954545454545457
Epoch 8: Character accuracy: 0.7744545454545455, Training accuracy: 0.3978846
153846154, Validation accuracy: 0.38181818181818183
Epoch 9: Character accuracy: 0.7774545454545455, Training accuracy: 0.40625,
Validation accuracy: 0.3881818181818182
Epoch 10: Character accuracy: 0.7785454545454545, Training accuracy: 0.412403
8461538462, Validation accuracy: 0.3936363636363636
Epoch 11: Character accuracy: 0.7813636363636364, Training accuracy: 0.420192
3076923077, Validation accuracy: 0.3986363636363636
Epoch 12: Character accuracy: 0.7835454545454545, Training accuracy: 0.425673
07692307693, Validation accuracy: 0.4
Epoch 13: Character accuracy: 0.7841818181818182, Training accuracy: 0.432211
5384615385, Validation accuracy: 0.3990909090909091
Epoch 14: Character accuracy: 0.786, Training accuracy: 0.4355769230769231, V
alidation accuracy: 0.4036363636363636
Epoch 15: Character accuracy: 0.7872727272727272, Training accuracy: 0.441153
8461538462, Validation accuracy: 0.4090909090909091
Epoch 16: Character accuracy: 0.7886363636363637, Training accuracy: 0.448942
3076923077, Validation accuracy: 0.41409090909090907
Epoch 17: Character accuracy: 0.7895454545454546, Training accuracy: 0.451442
3076923077, Validation accuracy: 0.41545454545454547
Epoch 18: Character accuracy: 0.7895454545454546, Training accuracy: 0.455865
3846153846, Validation accuracy: 0.41545454545454547
Epoch 19: Character accuracy: 0.7894545454545454, Training accuracy: 0.458653
84615384613, Validation accuracy: 0.415
Epoch 20: Character accuracy: 0.7883636363636364, Training accuracy: 0.463365
3846153846, Validation accuracy: 0.4109090909090909
Epoch 21: Character accuracy: 0.7885454545454545, Training accuracy: 0.468269
23076923077, Validation accuracy: 0.41363636363636364
Epoch 22: Character accuracy: 0.7896363636363637, Training accuracy: 0.47125,
Validation accuracy: 0.41454545454545455
Epoch 23: Character accuracy: 0.7906363636363636, Training accuracy: 0.479423
0769230769, Validation accuracy: 0.41818181818181815
Epoch 24: Character accuracy: 0.7922727272727272, Training accuracy: 0.485, V
alidation accuracy: 0.42
Epoch 25: Character accuracy: 0.7915454545454546, Training accuracy: 0.490192
3076923077, Validation accuracy: 0.41954545454545455
Epoch 26: Character accuracy: 0.7916363636363636, Training accuracy: 0.492980
7692307692, Validation accuracy: 0.41863636363636364
Epoch 27: Character accuracy: 0.7927272727272727, Training accuracy: 0.496730
76923076925, Validation accuracy: 0.42363636363636364
Epoch 28: Character accuracy: 0.7930909090909091, Training accuracy: 0.500576
923076923, Validation accuracy: 0.42363636363636364
Epoch 29: Character accuracy: 0.7932727272727272, Training accuracy: 0.505576
```

```
923076923, Validation accuracy: 0.4240909090909091
Epoch 30: Character accuracy: 0.7940909090909091, Training accuracy: 0.50875,
Validation accuracy: 0.425
```





Part 6: End-to-End System

The end-to-end system takes in a single CAPTCHA image and attempts to decode it. Currently, the end-to-end system performs character classification using the "best" CNN model from training as it yielded the highest accuracies without overfitting. The end-to-end system outputs its prediction of each character (5 total) and the input itself for comparison.

In [96]:
```
def decodeCharacter (encodedValue):
  if (encodedValue < 10):
    return str(encodedValue)
  else:
    return chr(encodedValue + 55)
```

In [183]:
```
def e2emodel (imgs):
  model = CaptchaCNN()
  model.load_state_dict(torch.load(get_model_name(model.name, epoch=20, learni
ng_rate=0.001)))

  # Prediction
  out = model(imgs)
  pred = out.max(1, keepdim=True)[1].squeeze(1).tolist()[:5]
  CAPTCHA_prediction = list(map(decodeCharacter, pred))
  print(f"Predicted output = {CAPTCHA_prediction}")

  # Plot batch of images
  plt.imshow(imgs[0][0])
```

In [184]:
```
imgs, labels = next(iter(test))
e2emodel(imgs)
```

```
Predicted output = ['L', 'V', 'E', 'Z', 'H']
```