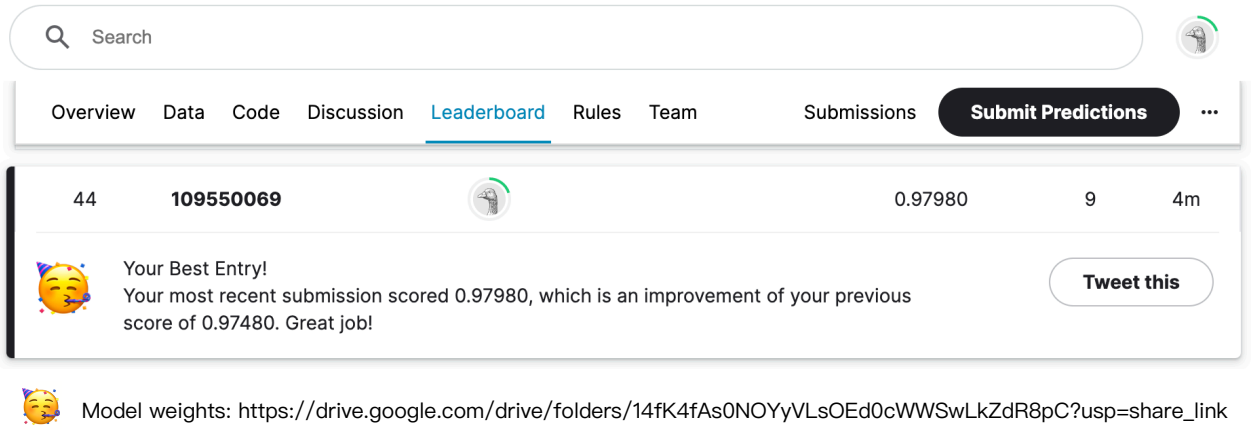
109550069 林宇柔

HW5 Report

Result (Public Leaderboard)



Model weights: https://drive.google.com/drive/folders/14fK4fAs0NOYyVLsOEd0cWWSwLkZdR8pC?usp=share_link

Environment Details

```
[1] from google.colab import drive
    drive.mount('/content/drive')
```

I finished this assignment in Google Colab, while putting all the data used on my Google Drive, I had to implement the code above to get them. Also, I mostly ran it on GPU as my hardware accelerator so the program could be faster (But I often run out of GPU so I switched between 3 Google accounts.)

Implementation Details – HW5_train.ipynb

First I imported all the modules that will be used and declared the paths to get the data, also, set the device to GPU (sometimes I change it to “cpu” if I run out of GPU for all 3 Google accounts of mine which makes me frustrated)

```
[2] import csv
    import cv2
    import numpy as np
    import random
    import os

    from tqdm import tqdm

    import torch
    import torch.nn as nn
    from torch.utils.data import Dataset, DataLoader
    import torch.nn.functional as F
    from torchvision.transforms.functional import to_tensor

    from collections import OrderedDict
```

```
[3] TRAIN_PATH = "/content/drive/MyDrive/hw5/captcha-hacker/train"
    TEST_PATH = "/content/drive/MyDrive/hw5/captcha-hacker/test"
    device = "cuda"
```

Then I declared a string including all the labels that might be predicted

```
[4] characters = '0123456789abcdefghijklmnopqrstuvwxyz'
```

Next I tried to extract the data, the root is to decide if taking the TEST_PATH or the TRAIN_PATH, the data is a list of data from different tasks, the digit and the count are for calculating CTC loss, which are respectively the input_length and the target_length. By the string mentioned above, I turned the labels into the number of its index so the string can be used as a dictionary. The data has to be transformed into tensors, so except for the image which is turned into torch.float after being read with cv2, the variables are turned into torch.long.

```
[38] class taskData(Dataset):
       def __init__(self, data, root, digit, count):
         self.root = root
         self.data = data
         self.digit = digit
         self.count = count

       def __getitem__(self, index):
         filename, label = self.data[index]
         # turn everything to tensor
         img = to_tensor(cv2.imread(f"{self.root}/{filename}")) # the image
         encode = torch.tensor([characters.find(x) for x in label], dtype=torch.long) # turn characters to indices (target)
         digits = torch.full(size=(1, ), fill_value=self.digit, dtype=torch.long) # how many digits originally predicted (inputLength)
         counts = torch.full(size=(1, ), fill_value=self.count, dtype=torch.long) # how many digits should there be (targetLength)
         return img, encode, digits, counts

       def __len__(self):
         return len(self.data)
```

To separate the data by the task, I read the csv file by rows and identify their beginnings to decide which list should they append to. Using the dataset above, set the roots to the training path and the count to be 1, 2, and 4 (the digits to be predicted in the 3 tasks), and set the digit to 3, 5, and 9 since the input length would be at least 2n+1 in the worst case, and I changed them to be smaller number to fit the actual input. Then I split the data into 9:1 for training and validation, and put them into the dataLoader.

```
[6]  train_data1 = []
     train_data2 = []
     train_data3 = []

     # append the data classified by starting string
     with open(f'{TRAIN_PATH}/annotations.csv', newline='') as csvfile:
       for row in csv.reader(csvfile, delimiter=','):
         if row[0].startswith("task1"):
           train_data1.append(row)
         elif row[0].startswith("task2"):
           train_data2.append(row)
         elif row[0].startswith("task3"):
           train_data3.append(row)

     ds1 = taskData(train_data1, root=TRAIN_PATH, digit=3, count=1)
     train1_ds, val1_ds = torch.utils.data.random_split(ds1, [1800, 200])
     train1_dl = DataLoader(train1_ds, batch_size=300, num_workers=2, drop_last=True, shuffle=True)
     val1_dl = DataLoader(val1_ds, batch_size=200, num_workers=2, drop_last=True, shuffle=True)

     ds2 = taskData(train_data2, root=TRAIN_PATH, digit=4, count=2)
     train2_ds, val2_ds = torch.utils.data.random_split(ds2, [2250, 250])
     train2_dl = DataLoader(train2_ds, batch_size=300, num_workers=4, drop_last=True, shuffle=True)
     val2_dl = DataLoader(val2_ds, batch_size=250, num_workers=4, drop_last=True, shuffle=True)

     ds3 = taskData(train_data3, root=TRAIN_PATH, digit=6, count=4)
     train3_ds, val3_ds = torch.utils.data.random_split(ds3, [2700, 300])
     train3_dl = DataLoader(train3_ds, batch_size=300, num_workers=4, drop_last=True, shuffle=True)
     val3_dl = DataLoader(val3_ds, batch_size=200, num_workers=4, drop_last=True, shuffle=True)
```

As for the model, I found a pre-trained model online and made some changes on my own. It consists of a sequence with 5 repeating blocks, a dropout layer, a LSTM layer and finally a linear layer. In the repeating blocks, a convolutional layer has an input channel at lower layer and an output layer at higher layer ranging from 3 to 256, a batch normalization layer to flatten the height and the width, a activation layer to turn complex data into simple functions and make it non-linear, also a pooling layer after every 2 subsequences of the former 3

layers for inputs containing various planes. After the sequence the input would be reshaped and permute the data positions before putting in the LSTM layer and linear layer. Another thing to notice is that the output feature of the linear layer should be 36 as there are 36 characters in total.

```
[ ] class Model(nn.Module):
      def __init__(self): # the output should be limited from 0 to z
        super(Model, self).__init__()
        # channels and pools will be different in different layers
        channels = [32, 64, 128, 256, 256]
        pools = [2, 2, 2, 2, (2, 1)]
        modules = OrderedDict()
        # the block would be iterated in the sequence ((conv2d, batchnorm, relu)*2, maxpool)
        def blocks(name, in_channels, out_channels):
            modules[f'conv{name}'] = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=(1, 1))
            modules[f'batchnorm{name}'] = nn.BatchNorm2d(out_channels)
            modules[f'relu{name}'] = nn.ReLU(inplace=True)
        channel1 = 3 # (3, 32 -> 32, 32)
        for block, (n_channel, k_pool) in enumerate(zip(channels, pools)):
            for layer in range(1, 3):
                blocks(f'{block+1}{layer}', channel1, n_channel)
                channel1 = n_channel
            modules[f'maxpool{block + 1}'] = nn.MaxPool2d(k_pool)
        # dropout at last
        modules[f'dropout'] = nn.Dropout(0.25, inplace=True)
        self.cnn = nn.Sequential(modules)
        # lstm and the last linear layer
        self.lstm = nn.LSTM(input_size=512, hidden_size=128, num_layers=2, bidirectional=True)
        self.final = nn.Linear(in_features=256, out_features=36)

      def forward(self, x):
        x = self.cnn(x)
        x = x.reshape(x.shape[0], -1, x.shape[-1])
        x = x.permute(2, 0, 1)
        x, _ = self.lstm(x)
        x = self.final(x)
        return x
```

In the training process I used the tqdm function on the data loader so I can track the training process, for all the tensor I run them on GPU, and calculate the CTC loss of the predictions, then update for the next epoch.

```
[ ] def train(model, optimizer, dataloader):
      model.train()
      # track the process
      with tqdm(dataloader) as pbar:

        # run through all data
        for batch_index, (data, encode, digits, counts) in enumerate(pbar):
          # to cuda
          data = data.to(device)
          encode = encode.to(device)
          # predict
          pred = model(data)
          # calculate the loss
          outputLs = F.log_softmax(pred, dim=-1)
          loss = F.ctc_loss(outputLs, encode, digits, counts)
          # update
          optimizer.zero_grad()
          loss.backward()
          optimizer.step()

          pbar.set_description(f'Training Progress')
```

In a similar way, I calculate the accuracy of the label and the prediction of each validation.

```
[ ] def valid(model, optimizer, dataloader, num):
      model.eval()
      # track the process
      with tqdm(dataloader) as pbar, torch.no_grad():

        # run through all data
        for batch_index, (data, encode, digits, counts) in enumerate(pbar):
          # to cuda
          data = data.to(device)
          encode = encode.to(device)
          # predict and calculate the accuracy
          pred = model(data)
          acc = accuracy(encode, pred, num)

          print('Accuracy: %.3f' % acc)
```

To calculate the accuracy, first transform both the labels and the predictions into strings of characters with the encoded ones as the index, then check which task is it to get prediction with corresponding digit counts, and calculate the number of correct predictions.

```
[ ] def accuracy(encode, output, num):
      outputs = output.detach().permute(1, 0, 2).argmax(dim=-1).cpu().numpy()
      encoded = encode.cpu().numpy()
      correct = 0
      total = 0
      for true, pred in zip(encoded, outputs):
        total += 1
        goal = ''.join([characters[x] for x in true]).replace(' ', '')
        prediction = ''.join([characters[x] for x in pred])
        print(goal , prediction, p(prediction, num)) # check the result
        if goal == p(prediction, num):
          correct += 1
      acc = correct / total
      return acc
```

Below is the function to determine the output, according to the prediction I printed, task 1 results would be the the first of the array consisting of 4 digits, the first and the last for task 2. As for task three, there will be 6 digits, the first and the last will be correct, but the places in between might be 0 which means blanks are predicted (which is wrong), and sometimes there will be unnecessary duplicate predictions of a digit, so I set up conditions to deal with these problems.

```
[ ] def p(seq, num):
      # TASK1
      if num == 1:
        return seq[0]
      # TASK2
      elif num == 2:
        return seq[0]+seq[3]
      # TASK3
      else:
        s = ''.join([x for j, x in enumerate(seq[:-1]) if x != characters[0] and j != 0])
        if len(s) < 2 and seq[3] == '0' and seq[4] == '0':
          s += '0'
        if len(s) < 2 and seq[1] == '0' and seq[2] == '0':
          s = '0' + s
        s = seq[0] + s
        if len(s) == 4:
          if s[2] != s[3]:
            s = "".join(sorted(set(s), key=s.index))
          else:
            s = s[:3]
        s += seq[-1]
        return s
```

Start training one model for each tasks, if the performance it not good enough, I go through more epochs with smaller learning rates.

```
[ ] # TASK1 lr=1e-3
    model1 = Model().to(device)
    optimizer = torch.optim.Adam(model1.parameters(), 1e-3, amsgrad=True)
    epochs = 50
    for epoch in range(1, epochs + 1):
      print("Epoch ", epoch)
      train(model1, optimizer, train1_dl)
      valid(model1, optimizer, val1_dl, 1)
      print("-------------------------------------------------------------------------------")
```

After training I save the model weights.

```
[ ] torch.save(model1, 'model1.pth')
    !cp model1.pth ./drive/MyDrive/hw5/captcha-hacker/model1.pth
```

Implementation Details – HW5_inference.ipynb

The file is mostly the same as the training file, except the function getting the dataset and the one to write a csv file.

```
[ ] class taskData(Dataset):
      def __init__(self, data, root):
        self.root = root
        self.data = data

      def __getitem__(self, index):
        filename, label = self.data[index]
        img = to_tensor(cv2.imread(f"{self.root}/{filename}"))
        return img, filename

      def __len__(self):
        return len(self.data)
```

This time I only extract the images and their filenames from the data set.

```
[ ] model1 = Model().to(device)
    model1 = torch.load('model1.pth')
    model1.eval()

    with open('submission.csv', 'w', newline='') as csvfile:
      csv_writer = csv.writer(csvfile, delimiter=',')
      csv_writer.writerow(["filename", "label"])
      for image, filenames in ds1:
        output = model1(image.unsqueeze(0).to(device))
        outputs = output.detach().permute(1, 0, 2).argmax(dim=-1)
        csv_writer.writerow([filenames, decode(outputs[0], 1)])
```

To predict the labels of the test data, I create a model running on GPU and load the model weights that I trained, turn it into evaluation mode. Then create a csv file that includes all the filenames and the decoded outputs predicted by the model. The decoding function is the same a “p” function in the training file.

```
[ ] !cp submission.csv ./drive/MyDrive/hw5/captcha-hacker/submission.csv
```

Finally save the csv file to my drive.