Вступительный экзамен DL Advanced Весна'23. Программирование

Данные взяты отсюда.

При решении задания в качестве источника использовался следующий туториал. Подход, реализованный в данной работе, основан на раздельном распознавании сокращенного названия провинции (первого иероглифа в номере) и оставшейся части номера, состоящей из заглавных латинских букв и цифр.

В данной работе подсчитыватся только доля правильных ответов по словам и по символам. CER не считалось, так как количество предсказываемых символов фиксированно и равно 7.

In [3]:
```
import numpy as np
import zipfile
import cv2
import os
import io
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, Dataset
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import optim
from torch.nn.utils.clip_grad import clip_grad_norm_
from torchvision import transforms as T
from tqdm.notebook import tqdm
from matplotlib import pyplot as plt

import random
from datetime import timedelta
import time
from sklearn.metrics import accuracy_score
import yaml
```

In [15]:
```
with open("cfg.yaml", "r", encoding="utf-8") as f:
    cfg = yaml.load(f, Loader=yaml.FullLoader)


DIR = cfg["data_path"]
if DIR.endswith(".zip"):
    DIR = DIR[:-4]
```

In [5]:
```
# Разархивирование данных

with zipfile.ZipFile(f"{DIR}.zip", 'r') as zf:
    zf.extractall("")

# Выделениие валидационной выборки

os.mkdir(f"{DIR}/val/")
train_filenames = os.listdir(f"{DIR}/train")

train_filenames, val_filenames = train_test_split(train_filenames, test_size=0.15, shuff

for filename in val_filenames:
    os.rename(f"{DIR}/train/{filename}", f"{DIR}/val/{filename}")

# Сжатие полученных трех выборок
```

```python
with zipfile.ZipFile(f"{DIR}.zip", "w") as zf:
    for dirname, subdirs, files in os.walk(f"{DIR}"):
        zf.write(dirname)
        for filename in tqdm(files):
            zf.write(os.path.join(dirname, filename))

# Удаление разархивированного файла

for split in "train", "val", "test":
    for filename in os.listdir(f"{DIR}/{split}"):
        os.remove(f"{DIR}/{split}/{filename}")
    os.rmdir(f"{DIR}/{split}")
os.rmdir(DIR)
```

In [6]:

```python
tfms = T.Compose([
    T.ToTensor(),
    T.ConvertImageDtype(torch.uint8),
    T.CenterCrop((cfg["preprocess"]["img_height"], cfg["preprocess"]["img_width"])),
])
```

In [7]:

```python
# Функция для выделение метки из названия файла и перевод в правильную кодировку

def label_filter(name):
    return name[name.find("-", 9) + 1:-4].encode('cp437').decode('utf-8')
```

In [8]:

```python
provinces = cfg["provinces"]  # список всех сокращенных названий провинций Китая

vocab = [chr(idx) for idx in list(range(ord("A"), ord("Z") + 1)) + list(range(ord("0"),
```

In [9]:

```python
province_map = {province: idx for idx, province in enumerate(provinces)}
symbol_map = {symbol: idx for idx, symbol in enumerate(vocab)}

province_map_rev = {idx: province for idx, province in enumerate(provinces)}
symbol_map_rev = {idx: symbol for idx, symbol in enumerate(vocab)}
```

In [10]:

```python
# Класс для чтения данных из архивированного файла

class ZipDataset(Dataset):
    def __init__(self, path, label_filter=None, prefix="", transform=None):
        f = open(path, 'rb')
        self.zip_content = f.read()
        f.close()
        self.zip_file = zipfile.ZipFile(io.BytesIO(self.zip_content), 'r')
        self.label_filter = label_filter
        self.prefix = prefix
        self.name_list = list(filter(lambda filename: filename.endswith(".jpg") and file
                                     self.zip_file.namelist()))
        self.transform = transform

    def __getitem__(self, key):
        name = self.name_list[key]
        buf = self.zip_file.read(name=name)
        img = cv2.imdecode(np.frombuffer(buf, dtype=np.uint8), cv2.IMREAD_GRAYSCALE)
        if self.transform is not None:
            img = self.transform(img)
        if self.label_filter:
            name = label_filter(name)
        return img, name

#     def collate_fn(self, key):

    # def convert_label(self, name):
    #     if self.label_filter:
    #         name = label_filter(name)
```

```python
    #     return province_map[name[0]], [symbol_map[symbol] for symbol in name[1:]]

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

In [11]:
```python
batch_size = cfg["training"]["batch_size"]

train_dataset = ZipDataset(cfg["data_path"], prefix=f"{DIR}/train", label_filter=label_f
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

val_dataset = ZipDataset(cfg["data_path"], prefix=f"{DIR}/val", label_filter=label_filte
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)

test_dataset = ZipDataset(cfg["data_path"], prefix=f"{DIR}/test", label_filter=label_fil
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

В основе реализованной в данной работе модели лежит модель из туториала. Изменения:

```
- Только один слой двунаправленной LSTM
- Добавлены полносвязные слои для сжатия по ширине и высоте
- Выход рекуррентного слоя разбивается на два. Первый выход преобразуется в
распределение над словарем названий провинций, второй - в распределение на
словарем латинских заглавных букв и цифр
```

In [43]:
```python
class BidirectionalLSTM(nn.Module):

    def __init__(self, nIn, nHidden, nOut):
        super(BidirectionalLSTM, self).__init__()
        self.rnn = nn.LSTM(nIn, nHidden, bidirectional=True)
        self.embedding_1 = nn.Linear(nHidden * 2, nOut[0])
        self.embedding_2 = nn.Linear(nHidden * 2, nOut[1])

    def forward(self, input):
        self.rnn.flatten_parameters()
        recurrent, _ = self.rnn(input)
        recurrent = [recurrent[0], recurrent[1:]]
        t, b, h = recurrent[1].size()
        t_rec = recurrent[1].view(t * b, h)
        output = [self.embedding_1(recurrent[0]).view(1, b, -1), self.embedding_2(t_rec)
        return output

class CRNN(nn.Module):

    def __init__(self, opt, leakyRelu=False):
        super(CRNN, self).__init__()

        ks = [3, 3, 3, 3, 3, 3, 2]
        ps = [1, 1, 1, 1, 1, 1, 0]
        ss = [1, 1, 1, 1, 1, 1, 1]
        nm = [64, 128, 256, 256, 512, 512, 512]

        cnn = nn.Sequential()

        def convRelu(i, batchNormalization=False):
            nIn = opt['nChannels'] if i == 0 else nm[i - 1]
            nOut = nm[i]
            cnn.add_module('conv{0}'.format(i),
                           nn.Conv2d(nIn, nOut, ks[i], ss[i], ps[i]))
            if batchNormalization:
                cnn.add_module('batchnorm{0}'.format(i), nn.BatchNorm2d(nOut))
            if leakyRelu:
                cnn.add_module('relu{0}'.format(i),
                               nn.LeakyReLU(0.2, inplace=True))
```

```python
            else:
                cnn.add_module('relu{0}'.format(i), nn.ReLU(True))

        convRelu(0)
        cnn.add_module('pooling{0}'.format(0), nn.MaxPool2d((2, 2), 2))
        convRelu(1)
        cnn.add_module('pooling{0}'.format(1), nn.MaxPool2d((2, 2), 2))
        convRelu(2, True)
        convRelu(3)
        cnn.add_module('pooling{0}'.format(2),
                       nn.MaxPool2d((2, 2), (2, 1), (0, 1)))
        convRelu(4, True)
        convRelu(5)
        cnn.add_module('pooling{0}'.format(3),
                       nn.MaxPool2d((2, 2), (2, 1), (0, 1)))
        convRelu(6, True)

        self.cnn = cnn

        self.rnn = BidirectionalLSTM(opt['nHidden']*2, opt['nHidden'], opt['nClasses'])
        self.linear_h = nn.Linear(7, 1)
        self.linear_w = nn.Linear(101, 7)

    def forward(self, input):
        conv = self.cnn(input)
        conv = self.linear_w(conv)
        conv = conv.permute(0, 1, 3, 2)
        conv = self.linear_h(conv)
        conv = conv.permute(0, 1, 3, 2)
        conv = conv.squeeze(2)
        conv = conv.permute(2, 0, 1)
        output = self.rnn(conv)
        output[0] = output[0].squeeze()
        output[1] = output[1].transpose(1,0)
        return output
```

In [17]:
```python
device = torch.device(cfg["training"]["device"])
model = CRNN(cfg["model"]).to(device)
num_criterion = nn.CTCLoss(reduction="mean", zero_infinity=True)
province_criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=cfg["training"]["lr"])
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=cfg["training"]["epoch
```

In [28]:
```python
y_val = torch.Tensor().to(dtype=torch.int8)
for batch in tqdm(val_dataloader):
    b_labels = torch.cat(
    (
        torch.LongTensor([province_map[batch[1][i][0]] for i in range(len(batch[1]))]).v
        torch.LongTensor([[symbol_map[symbol] for symbol in batch[1][i][1:]] for i in ra
    ), dim=1)
    y_val = torch.cat((y_val, b_labels), dim=0)
```

```
0%|          | 0/313 [00:00<?, ?it/s]
```

In [29]:
```python
y_test = torch.Tensor().to(dtype=torch.int8)
for batch in tqdm(test_dataloader):
    b_labels = torch.cat(
    (
        torch.LongTensor([province_map[batch[1][i][0]] for i in range(len(batch[1]))]).v
        torch.LongTensor([[symbol_map[symbol] for symbol in batch[1][i][1:]] for i in ra
    ), dim=1)
    y_test = torch.cat((y_test, b_labels), dim=0)
```

```
0%|          | 0/105 [00:00<?, ?it/s]
```

In [ ]:

```
seed = cfg["training"]["seed"]

random.seed = (seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
model.cuda()

train_losses = []
val_losses = []

for epoch in range(cfg["training"]["epochs"]):
    print("Training {} epoch".format(epoch + 1))
    start = time.time()
    mean_loss = 0
    model.train()
    for step, batch in enumerate(tqdm(train_dataloader)):
        # if (step + 1) % 200 == 0:
        #     duration = timedelta(seconds=int(time.time() - start))
        #     print('Batch {:>5,}  of  {:>5,}. Loss {:.3}  Time: {:}.'.format(step + 1,

        torch.cuda.empty_cache()
        b_input = batch[0].to(device)
        b_labels = torch.cat(
            (
                torch.LongTensor([province_map[batch[1][i][0]] for i in range(len(batch[
                torch.LongTensor([[symbol_map[symbol] for symbol in batch[1][i][1:]] for
            ), dim=1).to(device)
        model.zero_grad()

        province_logits, num_logits = model(b_input / 255)
        province_logits = F.softmax(province_logits, 1)
        num_logits = F.log_softmax(num_logits, 2).transpose(0, 1)
        pred_sizes = (torch.ones(len(batch[0])) * 7).long().to(device)

        loss = province_criterion(province_logits, b_labels[:, 0])
        loss += num_criterion(num_logits, b_labels[:, 1:], pred_sizes, pred_sizes)

        optimizer.zero_grad()
        loss.backward()

        max_grad_norm = 0.05
        clip_grad_norm_(model.parameters(), max_grad_norm)
        optimizer.step()
        mean_loss += loss.item()

    scheduler.step()
    mean_loss = mean_loss / len(train_dataloader)

    train_losses.append(mean_loss)
    print("Mean loss: " , mean_loss)
    print("Training epoch took:" , timedelta(seconds=int(time.time() - start)))
    torch.save(model, cfg["model_path"])
    print()
    print("Validation:")
    model.eval()

    start = time.time()
    predictions = torch.Tensor().to(dtype=torch.int8)
    val_loss = 0

    for batch in tqdm(val_dataloader):

        b_input = batch[0].to(device)
        b_labels = torch.cat(
        (
```

```python
            torch.LongTensor([province_map[batch[1][i][0]] for i in range(len(batch[1]))
            torch.LongTensor([[symbol_map[symbol] for symbol in batch[1][i][1:]] for i i
        ), dim=1).to(device)

        with torch.no_grad():
            province_logits, num_logits = model(b_input / 255)
            province_logits = F.softmax(province_logits, 1)
            num_logits = F.log_softmax(num_logits, 2).transpose(0, 1)
            pred_sizes = (torch.ones(len(batch[0])) * 7).long().to(device)

            loss = province_criterion(province_logits, b_labels[:, 0])
            loss += num_criterion(num_logits, b_labels[:, 1:], pred_sizes, pred_sizes)

        predictions = torch.cat((predictions, torch.cat((
            province_logits.argmax(dim=1).view(-1, 1).cpu().detach(),
            num_logits.transpose(0, 1).argmax(dim=-1).cpu().detach()
            ), dim=1)), dim=0)
        torch.cuda.empty_cache()

    print("Accuracy by word: {:4.2f}".format(np.equal(y_val, predictions).all(axis=1).fl
    print("Accuracy by char: {:4.2f}".format(np.equal(y_val, predictions).float().mean()
    val_losses.append(val_loss / len(val_dataloader))
    print("Validation took: {:}".format(timedelta(seconds = int(time.time() - start))))
    print()
```

In [46]:

```python
model = torch.load("model_weights.pt")
```

In [57]:

```python
print("Testing:")
model.eval()

start = time.time()
test_predictions = torch.Tensor().to(dtype=torch.int8)

for batch in tqdm(test_dataloader):

    b_input = batch[0].to(device)
    b_labels = torch.cat(
    (
        torch.LongTensor([province_map[batch[1][i][0]] for i in range(len(batch[1]))]).v
        torch.LongTensor([[symbol_map[symbol] for symbol in batch[1][i][1:]] for i in ra
    ), dim=1).to(device)

    with torch.no_grad():
        province_logits, num_logits = model(b_input / 255)
        province_logits = F.softmax(province_logits, 1)
        num_logits = F.log_softmax(num_logits, 2).transpose(0, 1)

    test_predictions = torch.cat((test_predictions, torch.cat((
        province_logits.argmax(dim=1).view(-1, 1).cpu().detach(),
        num_logits.transpose(0, 1).argmax(dim=-1).cpu().detach()
        ), dim=1)), dim=0)
    torch.cuda.empty_cache()

print("Accuracy by word: {:4.2f}".format(np.equal(y_test, test_predictions).all(axis=1).
print("Accuracy by char: {:4.2f}".format(np.equal(y_test, test_predictions).float().mean
print("Testing took: {:}".format(timedelta(seconds = int(time.time() - start))))
print()
```

```
Testing:
  0%|          | 0/105 [00:00<?, ?it/s]
Accuracy by word: 0.89
Accuracy by char: 0.98
Testing took: 0:00:43
```

In [ ]: