EE596 Autumn

Final Report

Font Recognition Through Deep Learning

Diman Todorov

December 14, 2020

Contents

1. Objective

The objective for this project is to train a model which recognizes up to 10 different fonts. The problem statement of this project is:

Given an image of rendered text, classify the font used in the image.

Fonts come in font families comprised of sometimes dozens of members. For example, Helvetica has around 50 different versions for sale on myfonts.com

The text which will be considered is scoped to:
- English language
- English alphabet (lower- and upper-case)
- numbers 0-9
- include common punctuation characters
- written in black on a white background.
- written horizontally.
- the entire text in the studied image is rendered in a single font

Glyphs (characters representing more than one letter) will be included in the experiment.

The fonts included in the experiment are:

- Arial
- Comic Sans
- Times New Roman
- Courier New
- Calibri
- Candara
- Consolas
- Georgia
- Corbel
- Helvetica

Only the regular font weight will be considered. For example, in the Microsoft Typography Arial is comprised of 4 fonts: Arial, Arial Italic, Arial Bold and Arial Bold Italic. In this example only Arial will be included. Note that Arial Black is a separate font family.

2. Method

The project is conducted in the following phases:
1. Literature review
2. Data pre-processing
3. Image generation
4. Model training

The topic of font recognition is likely not very impactful as related literature is sparse and often of poor quality. For this reason, literature in the field of Optical Character Recognition (OCR) is included.

Data is generated using a Wikipedia database dump as source material. To keep the project feasible the reduced database dump is used which only contains abstracts of Wikipedia articles. The raw dump is offered in a Wikimedia specific XML format. Further, many abstracts are empty or become empty after sanitizing them to fit the project goal.

Once the data is stripped of metadata and special characters, it is used to generate PNG images for training and validating the model.

Finally, a model is trained on the synthetic data.

3. Literature Review

Although convolutional neural networks have been used for font recognition, the scope of application is narrow. For example, Tensmeyer et al. are interested in Arabic script [1]. The authors report very good classification performance of their method (above 90% accuracy) and they discuss challenges with the data. One such challenge is that the neural network seemed to be treating text darkness as a feature of the font. A method is proposed to augment the data in such a way that text darkness plays a lesser role when training the network. Further, the authors report what features the CNN learned as discriminators between classes.

Another interesting application of deep learning is reported by Bhunia et al. [2] who use a neural network to convert data from one font to another. Their approach is grouped into two components, a discriminator part, and a generator part. The discriminator is trained to classify the font of the input whereas the generator reproduces the text in the required font type. The authors claim that the innovation in their research is that their model can handle input of varying widths while previous work only copes with square images as input. The architecture of the network used in this research is a Conditional Generative Adversarial Network which is a specialized model for image translation. It takes an image as an input and produces an image at the output following certain rules which are learned while training the network.

Bychkov et al. [3] report on research proposing a model which to classifies fonts by slope, weight and type. The authors attempted to solve the problem using classical computer vision methods but found that there were limitations which were difficult to overcome. Once they applied convolutional neural networks (CNN) they arrived at a model with high classification accuracy (over 90%). Several different types of activation functions were compared, and the authors found that the ReLU function was most suitable for the problem. The authors started with the well-known VGG Net architecture however they propose a variation on the same as better suited for font classification.

Considering the paper [3] was written in 2020, it is confusing to read a report that VGG outperforms other networks in image classification. Although image captioning is a somewhat removed field, Atliha et al. [4] report that ResNet outperforms VGG for their decoder component. They compare 2 neural network architectures for suitability as the computer vision component in an automatic image captioning system. The two architectures are the popular convolutional neural networks VGG and ResNet. They find that ResNet outperforms VGG on the data by being more accurate but also faster to train. A notable conclusion of the paper is that in the problem of image captioning, changing just the encoder component plays a role in the overall accuracy. This is of relevance because in related literature, proposed models often differ

in their encoder as well as their decoder sections making it difficult to attribute differences in accuracy to a specific component of the model.

Although somewhat dated, a wide-spread model for OCR is proposed by Baoguang et al [5]. They name their model architecture a convolutional recurrent neural network (CRNN). It is specialized to work on sequences of text and is designed to be independent of script or language. The authors report success on mandarin characters and even musical notation.

4. Generating Data

The source material for generating the data are Wikipedia article abstracts. The database dump available on the Wikipedia website was dumped in March 2018 and contains abstracts for around 5.5 million articles. Out of those, the first 200 abstracts are used for a validation dataset and the next 4000 are used as a training dataset. All selected abstracts are rendered in each of the 10 fonts within the scope of the project resulting in a test dataset with 2000 images and a training dataset with 40,000 images.

The database dump is in an XML format which is around 650mb when compressed. This data was pre-processed to:

- Remove characters outside the scope of the project
- Remove empty abstracts
- Convert to a compressed text file with 1 abstract per line.

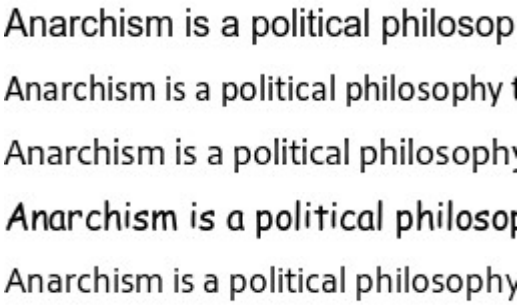
The full source code for the pre-processor is given in section 8.1

The images are stored on disk without compression and their filenames are formatted as: `"{number} {font}.png"`

Initially the images were rendered at 256x256px and a word wrapping algorithm was applied to ensure that the full image area contains interesting information. Three challenges were identified:

1. Many abstracts are so short that they fill only the first line with the rest of the image being white.
2. The applied word wrapping algorithm was prohibitively slow even when using a multi-threaded approach to generating data.
3. Many abstracts were longer than fits on a 256x256 image which made the word wrapping algorithm even slower.

To address these challenges, the data was re-generated in a resolution of 32x256 and text was cropped to the first 500 characters. The word wrapping algorithm was removed and only a single line is rendered in each image. The full source code for the image renderer is given in section 8.2. A custom DataLoader is implemented (see section 8.3) to allow feeding the data into PyTorch models. These are some example images:



5. Training Networks

In total 4 network architectures were considered in this project:

1. PyTorch tutorial network
2. VGG
3. GoogleNet
4. CRNN (ResNet)

The network from the PyTorch tutorial (see section 8.4) was used to validate that the project pipeline works as intended. Once it was verified that processing data works, it was not further considered. The VGG network would not fit in the GPU memory on my laptop and it was not further considered. The PyTorch implementation of GoogleNet would crash initially. Since CRNN gives very high accuracy on the dataset, GoogleNet was not considered for a comparison.

The CRNN implementation used in the project is from the public GitHub repository: https://github.com/bes-dev/crnn-pytorch with a minor change allowing it to process 3 channel PNGs rather than only a single channel.

When loading the data, it is normalized around a 3x0.5 mean and 3x0.5 standard deviation, this is to match the assumptions of the CRNN implementation. The data is shuffled and a random crop of size 32x32px is applied to introduce regularization. Very high accuracy is achieved within 3 training epochs using a mini-batch size of 100 observations.

Two interesting challenges were encountered while training the data. Initially, 256x256 images with single line abstracts would produce many purely white 32x32 squares. This skewed the experiment to the point where the networks would not reach better than 60% accuracy on the test set even after increasing the size of the synthetic dataset. This is addressed by reducing all data to single line images thus ensuring that all 32x32 squares contain text.

The second challenge was that the ImageMagick library used to render the images behaves oddly when a font-name cannot be found on the system. If a font cannot be found, the library always chooses Arial. Initially I had spelled “Comic Sans MS” as “Comic Sans” which produced 2 class labels with the same font type. This was uncovered only after modifying the training code to print class accuracy for each 1000 iterations of the training.

Two optimizers were compared, stochastic gradient descent and Adam, but no significant difference was found for this project. The full source code of the training module is given in section

6. Results and Discussion

Using the described approach, the network easily exceeds the accuracy expected in the project proposal:

```
   [3,   400] loss: 0.069 accuracy: 98.429
   class accuracy: Arial 98.6% Times New Roman 98.6% Courier New
98.6% Calibri 97.1% Candara 100% Georgia 100% Corbel 98.6%
Helvetica 92.9% Comic Sans MS 100% Garamond 100%
   Finished Training
```

After around 5-10 minutes of training on a laptop with a mobile class GPU, the model achieves almost perfect accuracy. Although further diagnostics could be conducted, such as

plotting the loss and accuracy over number of iterations, considering the good performance of the model in time, space and accuracy it seems like a moot point.

To make the project more interesting, a significant pivot in the problem statement would be required. One example would be to train the model to recognize fonts on real images such found in advertisements, street signage or magazine covers.

An open question after this investigation is why CRNN performed so well. The authors of the original method claim [5] that CRNN was designed specifically to recognize sequences of characters. Considering the size of the training images, 32x32px, sequences play no important role in this experiment. The specific implementation of CRNN used in this project uses ResNet as the backend for the CRNN. One possibility is that given the training data the algorithm degenerates to just ResNet.

7. References

1. Tensmeyer, C., D. Saunders, and T. Martinez. *Convolutional neural networks for font classification*. in *2017 14th IAPR international conference on document analysis and recognition (ICDAR)*. 2017. IEEE.
2. Bhunia, A.K., et al. *Word level font-to-font image translation using convolutional recurrent generative adversarial networks*. in *2018 24th International Conference on Pattern Recognition (ICPR)*. 2018. IEEE.
3. Bychkov, O., et al. *Using Neural Networks Application for the Font Recognition Task Solution*. in *2020 55th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*. 2020. IEEE.
4. Atliha, V. and D. Šešok. *Comparison of VGG and ResNet used as Encoders for Image Captioning*. in *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*. 2020. IEEE.
5. AN, T., *An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition.*

8. Appendix

8.1 Parse Wikipedia XML

```
import xml.sax
import gzip
import io
import re
import time

# 5.5 million abstracts ~ 10 minutes

class AbstractHandler(xml.sax.ContentHandler):
    def __init__(self, output):
        self.inAbstract = False
        self.abstract = ""
        self.output = output
        self.counter = 0

    def startElement(self, name, attrs):
        if name == "abstract":
```

```
            self.counter += 1
            self.inAbstract = True

    def endElement(self, name):
        if self.inAbstract and name == "abstract":
            sanitized = re.sub(r'[^a-zA-Z0-9+-:;()?! ]+', '', self.abstract)
            sanitized = re.sub(r' +', ' ', sanitized)
            if not self.abstract.startswith("|") and len(sanitized) > 25:
                self.output.write(sanitized + "\n")
            self.abstract = ""
            self.inAbstract = False

            if self.counter % 10000 == 0:
                print('processed %d abstracts. process time: %.3f' %
(self.counter, time.process_time()))

    def characters(self, content):
        if self.inAbstract:
            self.abstract += content

with gzip.open('enwiki-20180320-abstract.xml.gz', 'rb') as ingzip,
gzip.open("abstract-per-line.txt.gz", "wt", encoding="utf-8") as outgzip:
    parser = xml.sax.make_parser()
    parser.setFeature(xml.sax.handler.feature_namespaces, 0)
    parser.setContentHandler(AbstractHandler(outgzip))
    parser.parse(ingzip)
```

8.2 Abstract to PNG

```
import sys
import gzip
import time
from wand.color import Color
from wand.image import Image
from wand.drawing import Drawing
from wand.compat import nested
from textwrap import wrap
import threading

def word_wrap(
    image,
    ctx,
    text,
    roi_width,
    roi_height,
    ):
    """Break long text to multiple lines, and reduce point size
    until all text fits within a bounding box."""

    mutable_message = text
    iteration_attempts = 100
```

```
    def eval_metrics(txt):
        """Quick helper function to calculate width/height of text."""

        metrics = ctx.get_font_metrics(image, txt, True)
        return (metrics.text_width, metrics.text_height)

    while ctx.font_size > 0 and iteration_attempts:
        iteration_attempts -= 1
        (width, height) = eval_metrics(mutable_message)
        if height > roi_height:
            ctx.font_size -= 0.75  # Reduce pointsize
            mutable_message = text  # Restore original text
        elif width > roi_width:
            columns = len(mutable_message)
            while columns > 0:
                columns -= 1
                mutable_message = '\n'.join(wrap(mutable_message,
                        columns))
                (wrapped_width, _) = eval_metrics(mutable_message)
                if wrapped_width <= roi_width:
                    break
            if columns < 1:
                ctx.font_size -= 0.75  # Reduce pointsize
                mutable_message = text  # Restore original text
        else:
            break
    if iteration_attempts < 1:
        raise RuntimeError('Unable to calculate word_wrap for ' + text)
    return mutable_message

FONT_SIZE = 18

def render_text(fname, message, font):
    with Drawing() as draw:
        with Image(width=256, height=32, background=Color('white')) as \
            img:
            draw.font_family = font
            draw.font_size = FONT_SIZE
            draw.fill_color = Color('black')

            # mutable_message = word_wrap(img, draw, message, ROI_SIDE,
ROI_SIDE)

            draw.text(0, FONT_SIZE, message)
            draw(img)
            img.save(filename=fname)


N_IMAGES = 200

# FONTS = ["Arial", "Times New Roman", "Comic Sans", "Courier New",
"Calibri", "Candara", "Consolas", "Georgia", "Corbel", "Arial Black"]
```

```
FONTS = [
    'Arial',
    'Times New Roman',
    'Courier New',
    'Calibri',
    'Candara',
    'Georgia',
    'Corbel',
    'Helvetica',
    'Comic Sans MS',
    'Garamond',
    ]

total_count_lock = threading.Lock()
total_count = 0


def worker(lines, start, end):
    global total_count

    counter = start
    fname = 'test-data-1/{number} {font}.png'

    for i in range(start, end):

        for font in FONTS:
            render_text(fname.format(number=counter, font=font),
                        (lines[i])[:500], font)

        counter += 1

        total_count_lock.acquire()
        total_count += 1

        if total_count % 10 == 0:
            print 'processed: {images} time:
{time}'.format(images=total_count,
                    time=time.process_time())
        total_count_lock.release()


lines = []

with gzip.open('abstract-per-line.txt.gz', 'rt', encoding='utf-8') as \
    ingzip:
    lines = ingzip.readlines()

# threading.Thread(target=worker,args=(lines, 1000, 1999),).start()
# threading.Thread(target=worker,args=(lines, 2000, 2999),).start()
# threading.Thread(target=worker,args=(lines, 3000, 3999),).start()
# threading.Thread(target=worker,args=(lines, 4000, 4999),).start()

threading.Thread(target=worker, args=(lines, 0, 99)).start()
```

```
threading.Thread(target=worker, args=(lines, 100, 199)).start()
```

8.3 DataLoader

```
import os
import re
import torch
from torch.utils.data import Dataset, DataLoader
import cv2

def extract_font(fname):
    g = re.search(".+(\d+) ([a-zA-Z0-9 ]+).png", fname)
    return g.group(2)

class Abstract(object):

    def __init__(self, fonts, fname, transform):
        im = cv2.imread(fname)
        # crop top line. The reason is that a lot of the input data only has
one line
        # if we do random crop we will end up with a lot of just white data
        im = im[:32, :128]
        self.image = torch.from_numpy(im).permute(2, 0, 1).float()
        self.font = extract_font(fname)
        self.label = fonts.index(self.font)

    def getLabel(self):
        return self.label

    def getFont(self):
        return self.font

    def getImage(self):
        return self.image

def contains_any(string, fonts):
    actual = extract_font(string)

    return actual in fonts

class AbstractsDataset(Dataset):

    def __init__(self, directory, transform, fonts, max_count=0):
        self.entries = []
        self.directory = directory
        self.transform = transform

        i = 0

        for entry in os.scandir(self.directory):
            fname = entry.path
```

```
            if fname.endswith(".png") and entry.is_file() and
contains_any(fname, fonts):
                self.entries.append(Abstract(fonts, fname, transform))

            if max_count > 0 and i < max_count - 1:
                i += 1
            else:
                break

    def __len__(self):
        return len(self.entries)

    def __getitem__(self, idx):
        entry = self.entries[idx]

        label = entry.getLabel()
        image = entry.getImage()

        image = self.transform(image)

        return image, label

def transform_identity(obj):
    return obj

def test():
    d = AbstractsDataset("data-1", transform_identity, ["Arial"])
    dataloader = DataLoader(d, batch_size=4, shuffle=True, num_workers=1)

    for data in dataloader:
        images, labels = data

        print(labels)

# if __name__ == '__main__':
#     test()
```

8.4 PyTorch Tutorial Network

```
import torch.nn as nn
import torch.nn.functional as F

class TutorialNet(nn.Module):
    def __init__(self, num_classes=10):
        super(TutorialNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, num_classes)
```

```
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

8.5 CRNN

```
import torch
    import torch.nn as nn
    from torch.autograd import Variable

    import torchvision.models as models
    import string
    import numpy as np

    class CRNN(nn.Module):
        def __init__(self,
                     abc=string.digits,
                     backend='resnet18',
                     rnn_hidden_size=128,
                     rnn_num_layers=2,
                     rnn_dropout=0,
                     seq_proj=[0, 0]):
            super(CRNN, self).__init__()

            self.abc = abc
            self.num_classes = len(self.abc)

            self.feature_extractor = getattr(models,
backend)(pretrained=False)
            self.cnn = nn.Sequential(
                self.feature_extractor.conv1,
                self.feature_extractor.bn1,
                self.feature_extractor.relu,
                self.feature_extractor.maxpool,
                self.feature_extractor.layer1,
                self.feature_extractor.layer2,
                self.feature_extractor.layer3,
                self.feature_extractor.layer4
            )

            self.fully_conv = seq_proj[0] == 0
            if not self.fully_conv:
                self.proj = nn.Conv2d(seq_proj[0], seq_proj[1],
kernel_size=1)

            self.rnn_hidden_size = rnn_hidden_size
            self.rnn_num_layers = rnn_num_layers
            self.rnn = nn.GRU(self.get_block_size(self.cnn),
```

```
                            rnn_hidden_size, rnn_num_layers,
                            batch_first=False,
                            dropout=rnn_dropout, bidirectional=True)
            self.linear = nn.Linear(rnn_hidden_size * 2, self.num_classes)
            self.softmax = nn.Softmax(dim=2)

        def forward(self, x, decode=False):
            hidden = self.init_hidden(x.size(0),
next(self.parameters()).is_cuda)
            features = self.cnn(x)
            features = self.features_to_sequence(features)
            seq, hidden = self.rnn(features, hidden)
            seq = self.linear(seq)
            if not self.training:
                seq = self.softmax(seq)
                if decode:
                    seq = self.decode(seq)
            return seq.squeeze()

        def init_hidden(self, batch_size, gpu=False):
            h0 = Variable(torch.zeros( self.rnn_num_layers * 2,
                                batch_size,
                                self.rnn_hidden_size))
            if gpu:
                h0 = h0.cuda()
            return h0

        def features_to_sequence(self, features):
            b, c, h, w = features.size()
            assert h == 1, "the height of out must be 1"
            if not self.fully_conv:
                features = features.permute(0, 3, 2, 1)
                features = self.proj(features)
                features = features.permute(1, 0, 2, 3)
            else:
                features = features.permute(3, 0, 2, 1)
            features = features.squeeze(2)
            return features

        def get_block_size(self, layer):
            return layer[-1][-1].bn2.weight.size()[0]

        def pred_to_string(self, pred):
            seq = []
            for i in range(pred.shape[0]):
                label = np.argmax(pred[i])
                seq.append(label - 1)
            out = []
            for i in range(len(seq)):
                if len(out) == 0:
                    if seq[i] != -1:
                        out.append(seq[i])
                else:
```

```
                if seq[i] != -1 and seq[i] != seq[i - 1]:
                    out.append(seq[i])
            out = ''.join(self.abc[i] for i in out)
            return out

        def decode(self, pred):
            pred = pred.permute(1, 0, 2).cpu().data.numpy()
            seq = []
            for i in range(pred.shape[0]):
                seq.append(self.pred_to_string(pred[i]))
            return seq
```

8.6 Training

```
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

import crnn
import abstracts_dataset as A
import tutorial_net as Tutorial
from torch.autograd import Variable

# http://www.ironicsans.com/helvarialquiz/

fonts = [
      "Arial",
      "Times New Roman",
      "Courier New",
      "Calibri",
      "Candara",
      "Georgia",
      "Corbel",
      "Helvetica",
      "Comic Sans MS",
      "Garamond"]

def test_net(data, net):
    global fonts
    correct = 0
    total = 0
    with torch.no_grad():
        # images, labels = data
        images, labels = data[0].to(device), data[1].to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

```
        keys = labels.unique()
        class_accuracy = {}
        class_counts = {}

        for key in keys:
            class_accuracy[key.item()] = 0
            class_counts[key.item()] = 0
            for l in labels:
                if l.item() == key.item():
                    class_counts[key.item()] += 1


        for i in range(len(labels)):
            l = labels[i].item()
            if predicted[i] == l:
                class_accuracy[l] += 1

        for key in keys:
            class_accuracy[key.item()] /= class_counts[key.item()]

    return 100.0 * correct / total, class_accuracy

def train(trainloader, testloader, net, epochs):
    net.to(device)

    criterion = nn.CrossEntropyLoss()
    # optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
    optimizer = optim.Adam(net.parameters())

    for epoch in range(epochs):  # loop over the dataset multiple times
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # inputs, labels = data
            inputs, labels = data[0].to(device), data[1].to(device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)

            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
            if i % 100 == 99:    # print every 2000 mini-batches
                accuracy, class_accuracy = test_net(next(iter(testloader)),
net)
                print('[%d, %5d] loss: %.3f accuracy: %.3f' % (epoch + 1, i +
1, running_loss / 100, accuracy))
```

```
                castr = ""
                for i in range(len(fonts)):
                    castr += fonts[i] + " {0:.3g}".format(class_accuracy[i]*
100) + "% "
                print("class accuracy: " + castr)
                running_loss = 0.0

    print('Finished Training')

if __name__ == '__main__':
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    # Assuming that we are on a CUDA machine, this should print a CUDA
device:
    print(device)

    transform = transforms.Compose([
        transforms.RandomCrop(32),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    # FONTS = [
    #   "Arial",
    #   "Times New Roman",
    #   "Courier New",
    #   "Calibri",
    #   "Candara",
    #   "Consolas",
    #   "Georgia",
    #   "Corbel",
    #   "Arial Black"]

    print("Loading training data...")
    data1 = A.AbstractsDataset("data-1", transform, fonts, 100000)
    trainloader = torch.utils.data.DataLoader(data1, batch_size=100,
shuffle=True, num_workers=2)
    print("loaded %d images" % (data1.__len__()))

    print("Loading test data...")
    traindata1 = A.AbstractsDataset("test-data-1", transform, fonts, 1000)
    testloader = torch.utils.data.DataLoader(traindata1, batch_size=700,
shuffle=False, num_workers=2)
    print("loaded %d images" % (traindata1.__len__()))

    print("Training network...")

    # net = Tutorial.TutorialNet(num_classes=len(fonts))

    net = crnn.CRNN(fonts)

    train(trainloader, testloader, net, 3)
```