Deep Learning Engineering
Assignment 2:
HTR

Wilfred Van Casteren
nr:837377516
Open Universiteit
Masteropleiding AI

May 13, 2024

Abstract

This writing is mainly about handwritten text recognition. I have limited my main project to reading words using a convolutional recurrent neural network (CRNN), a basic RNN was used, as well as a long short-term memory (LSTM) and a gated recurrent unit (GRU). I have used the CTCLoss to optimize the sequence to sequence problem. The IAM dataset was used to provide images of words. For the experiments the words were limited to have a maximum size of 6 characters, also they contained only 15 characters (a-o). I didn't go all the way to fully optimize the training of the model.

1 Introduction

For this assignment I have tried two things firstly I have assured that it is possible to convert handwritten words to actual strings. I have used this site as a reference.

Secondly I have tried to implement Dive into deep learning's implementation of a single shot multibox detector. This took a lot of time without much result. So don't expect too much from this part.

I have used PyCharm to develop.

2 Word recognition

For this part of this assignment the IAM dataset has been used. This dataset contains handwritten data produced by asking multiple people to reproduce printed sentences. The final dataset is a collection of this data as a whole, or cut into words, sentences or lines.

The recognition of handwritten text in a word, sentence or line of text is a sequential problem. The sequence of handwritten characters needs to be converted into a sequence of unknown length, and consisting of an unknown number of characters, unknown at least for the network that needs to interpret the written words. Individual characters can be connected, or can be separated. Some of these differences are shown in Figure 1 and Figure 2.

Figure 1: A word from the IAM dataset.

Figure 2: Another word from the IAM dataset.

Adding to the complexity of recognizing words and characters, the images of the words themselves have all kinds of different shapes, no image is just as wide and high as any other.
I have limited the problem to the recognition of words, and not to lines, paragraphs or complete pages. I have used a convolutional recurrent network, in addition a ctc loss function will be used to find the optimal character combination.

2.1 Python packages

The main packages I have used are listed in Listing 1.

Listing 1: Packages used

```
python = "^3.12"
torch = "^2.2.1"
numpy = "^1.26.4"
matplotlib = "^3.8.3"
torchvision = "^0.17.1"
wakepy = "^0.7.2"
torchlens = "^0.1.12"
jiwer = "^3.0.4"
```

Wakepy is a package to keep a pc from going to sleep and torchlens is a package to visualize the model, jiwer is a package to calculate cer and other metrics. The other packages should already be known. The packages used were imported using the usual namespaces, so numpy is imported as np, ... I have limited myself to torch packages, and did not use packages belonging to other ecosystems.

2.2 Data

2.2.1 Create csv file

To get started the file containing all kinds of information concerning word images will serve as the source for a new csv-file containing the filenames (including its path) and their labels. Some string, split and concat operations need to be executed here to generate filenames including path names, also some erroneous and comment lines need to be checked for.

Listing 2: Reading word data

```
dataset = []
words = open(os.path.join(ascii_dir(), "words.txt"), "r").readlines()
for line in words:
 if line.startswith("#"):
  continue

 line_split = line.split(" ")
 if line_split[1] == "err":
  continue
```

```
 almost_file_name = line_split[0]
 file_name_parts = almost_file_name.split("-", 2)[:2]
 file_name = file_name_parts[0] + '/' + file_name_parts[0] + '-' + file_name_parts
    ↪ [1] + '/' + line_split[0] + '.png'
 label = line_split[-1].rstrip('\n')
 full_file_name = os.path.join(iam_dir(), "words", file_name)
 if not os.path.exists(full_file_name):
  print(f"File not found: {full_file_name}")
  continue

 dataset.append([full_file_name, label])

with open(generated_data_dir() + 'file_names-labels.csv', 'w') as file:
 writer = csv.writer(file, delimiter=',', quoting=csv.QUOTE_MINIMAL, lineterminator=
    ↪ '\n')
 writer.writerow(['file_name', 'label'])
 for item in dataset:
  writer.writerow([item[0], item[1]])
  file.flush()
file.close()
print('file created')
```

The resulting csv file looks like Figure 3. This file contains 90000+ lines.

file_name	label
../htr-torch-data/iam/words\a01/a01-000u/a01-000u-00-00.png	A
../htr-torch-data/iam/words\a01/a01-000u/a01-000u-00-01.png	MOVE
../htr-torch-data/iam/words\a01/a01-000u/a01-000u-00-02.png	to

Figure 3: Csv file containing image filenames and labels for the images.

2.2.2 Create dataloaders

The code to produce dataloaders looks like Listing 3. The parameters for the get_dataloaders-function contain an image transformation to convert the images to have the same dimensions, a char_to_int_map to be used to convert textual labels to numeric labels, num_of_rows to limit the dataloaders to contain a limited amount of records, text_label_max_length to limit the label length, char_set to check a label to contain only a limited amount of characters (a-o). Training and testing will be done using only images with labels no longer than 6 characters, also the characters used in the labels will contain all characters between a and o, including a and o, totaling 15 characters.

Listing 3: Dataloaders

```
def get_dataloaders(image_transform, char_to_int_map,int_to_char_map, num_of_rows,
    ↪ text_label_max_length, char_set):
 seq_dataset = HTRDataset('file_names-labels.csv', text_label_max_length,
    ↪ char_to_int_map, int_to_char_map, char_set, image_transform, num_of_rows)
 train_set, test_set = torch.utils.data.random_split(seq_dataset, [int(len(
    ↪ seq_dataset) * 0.8), int(len(seq_dataset) * 0.2)])

 train_loader = torch.utils.data.DataLoader(train_set, batch_size=4, shuffle=True)
 test_loader = torch.utils.data.DataLoader(test_set, batch_size=1, shuffle=True)

 return train_loader, test_loader
```

The images that will be used to train and test the neural network will look like in Figure 4 and Figure 5.

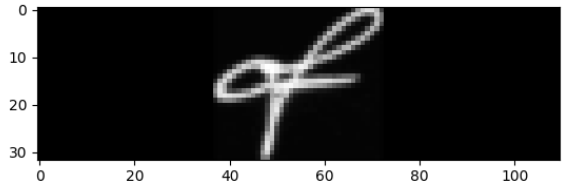

Figure 4: A padded and inverted word image.

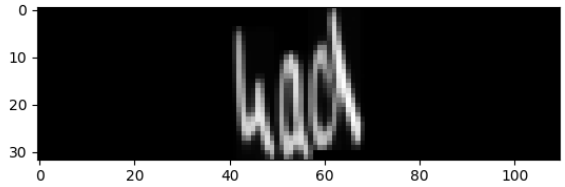

Figure 5: Another padded and inverted word image.

2.2.3 Dataset

To create a torch dataset, a subclass for the torch Dataset is made in Listing 4. Upon creation the dataset reads in the labels and images, it limits label length (line 12) and selects only labels that are made up a limited set of chars. It also limits the number of items in the dataset (line 30). The labels will need to be converted into numeric values, and they will need to be padded to be of the same length (line 17), images will need to be of the same height and width, greyscaled and inverted (line 19-20).

Listing 4: Subclassing Dataset

```
1  class HTRDataset(Dataset):
2   def __init__(self, file_name, text_label_max_length, char_to_int_map,
       ↪ int_to_char_map, char_set, image_transform, num_of_rows):
3    self.labels = torch.IntTensor()
4    self.images = torch.FloatTensor()
5    counter = 0
6    with open(generated_data_dir() + file_name, newline='') as file:
7     reader = csv.reader(file)
8     next(reader)
9     text_to_int = TextToInt(char_to_int_map)
10    fill_array = FillArray(length=text_label_max_length)
11    for row in reader:
12     if len(row[1]) > text_label_max_length:
13      continue
14     if not all_chars_in_set(row[1], char_set):
15      continue
16
17     lbl_tensor = torch.IntTensor(fill_array(text_to_int(row[1])))
18     img = read_image(row[0])
19     img = torchvision.transforms.functional.invert(img)
20     image = image_transform(img)
21
22     if image is None or lbl_tensor is None:
23      continue
24
25     self.images = torch.cat((self.images, image), 0)
26     self.labels = torch.cat((self.labels, lbl_tensor), 0)
27     counter = counter + 1
28     if counter == num_of_rows:
29      self.labels = self.labels.reshape([num_of_rows, text_label_max_length])
30      break
31   print('size images:',sys.getsizeof(self.images))
```

```
32
33  def __len__(self):
34   return len(self.labels)
35
36  def __getitem__(self, idx):
37   return self.images[idx], self.labels[idx]
```

2.2.4 Image transform

The transform (on line 20 of Listing 4) to make uniform images is to be seen in Listing 5.

Listing 5: Image transform

```
1 image_transform = v2.Compose(
2 [ResizeWithPad(h=32, w=110),
3 v2.Grayscale()
4 ])
```

The function to resize and pad an image is to be seen in Listing 6. The default w and h values in this case are overridden.

Listing 6: Resizing an image with padding

```
1  class ResizeWithPad:
2   def __init__(self, w=110, h=32):
3    self.w = w
4    self.h = h
5
6   def __call__(self, image):
7    sz = get_size(image)
8    w_1 = sz[0]
9    h_1 = sz[1]
10
11   ratio_f = self.w / self.h
12   ratio_1 = w_1 / h_1
13
14   if round(ratio_1, 2) != round(ratio_f, 2):
15    hp = int(w_1 / ratio_f - h_1)
16    wp = int(ratio_f * h_1 - w_1)
17    if hp > 0 and wp < 0:
18     hp = hp // 2
19     image = F.pad(image, (0, hp, 0, hp), 0, "constant")
20     return F.resize(image, [self.h, self.w])
21    elif hp < 0 and wp > 0:
22     wp = wp // 2
23     image = F.pad(image, (wp, 0, wp, 0), 0, "constant")
24     return F.resize(image, [self.h, self.w])
25   else:
26    return F.resize(image, [self.h, self.w])
```

2.3 Basic RNN-model

As previously said a convolutional recurrent neural network was used to handle the HTR-task. multiple sequences of convolutions and normalizations, reduce dimensions and increase channels. The final convolutional layer output will eventually be fed into a RNN layer (line 67). The reset_hidden function will reset the self.rnn_h layer to become a tensor of all zeroes. This function will be called for every new training batch (see subsubsection 2.6.5). It is important to note that the information for the basic RNN-model just as well applies to the LSTM-model and the GRU-model.

Listing 7: The RNN-based convolutional recurrent neural network

```
1 class CRNN_rnn(nn.Module):
```

```
def __init__(self):
 super(CRNN_rnn, self).__init__()

 self.num_classes = (16 + 1)
 self.image_H = 32

 self.conv1 = nn.Conv2d(1, 32, kernel_size=(3, 3))
 self.in1 = nn.InstanceNorm2d(32)

 self.conv2 = nn.Conv2d(32, 32, kernel_size=(3, 3))
 self.in2 = nn.InstanceNorm2d(32)

 self.conv3 = nn.Conv2d(32, 32, kernel_size=(3, 3), stride=2)
 self.in3 = nn.InstanceNorm2d(32)

 self.conv4 = nn.Conv2d(32, 64, kernel_size=(3, 3))
 self.in4 = nn.InstanceNorm2d(64)

 self.conv5 = nn.Conv2d(64, 64, kernel_size=(3, 3))
 self.in5 = nn.InstanceNorm2d(64)

 self.conv6 = nn.Conv2d(64, 64, kernel_size=(3, 3), stride=2)
 self.in6 = nn.InstanceNorm2d(64)

 self.postconv_height = 4
 self.postconv_width = 23

 self.rnn_input_size = self.postconv_height * 64
 self.rnn_hidden_size = 128
 self.rnn_num_layers = 2
 self.rnn_h = None
 self.rnn = nn.RNN(self.rnn_input_size, self.rnn_hidden_size, self.rnn_num_layers,
   ↪ batch_first=True)
 self.fc = nn.Linear(self.rnn_hidden_size, self.num_classes)

def forward(self, x):
 batch_size = x.shape[0]

 out = self.conv1(x)
 out = F.leaky_relu(out)
 out = self.in1(out)

 out = self.conv2(out)
 out = F.leaky_relu(out)
 out = self.in2(out)

 out = self.conv3(out)
 out = F.leaky_relu(out)
 out = self.in3(out)

 out = self.conv4(out)
 out = F.leaky_relu(out)
 out = self.in4(out)

 out = self.conv5(out)
 out = F.leaky_relu(out)
 out = self.in5(out)

 out = self.conv6(out)
 out = F.leaky_relu(out)
 out = self.in6(out)
```

```
64   out = out.permute(0, 3, 2, 1)
65   out = out.reshape(batch_size, -1, self.rnn_input_size)
66
67   out, rnn_h = self.rnn(out, self.rnn_h)
68   self.rnn_h = rnn_h.detach()
69   out = torch.stack([F.log_softmax(self.fc(out[i]), 1) for i in range(out.shape[0])
       ↪ ])
70   return out
71
72  def reset_hidden(self, batch_size):
73   h = torch.zeros(self.rnn_num_layers, batch_size, self.rnn_hidden_size)
74   self.rnn_h = Variable(h)
```

2.4 GRU-model

In this model, instead of using a simple RNN, a GRU-unit (a gated recurrent unit) is used. The GRU-unit is used in a RNN to handle sequence to sequence problems, just like the RNN. In the model used here every GRU-unit will be used bi-directionalline see line 35 of Listing 8, there will be two layers, and the gru_input_size depends on height off the featuremap and the number of channels after all convolutions are applied. The width of the featuremaps isn't used to calculate the number of input features for the GRU. In the forward function, on line 66 the output of line 64 is being permuted and reshaped (line 67) to become a tensor of size([4, 23, 256]). Where 4 is the batch size, 23 the feature width and 256 equals 4 times 64 (height times number of channels). The permuted and reshaped output after convolution is being fed into the GRU-object, together with the initial or intermediate self.gru_h values, self.gru_h is a tensor of size([4, 4, 128]), 4 for the number of layers times 2 when bi-directional, 4 for the batch size and 128 for the size of the GRU-object's hidden state. Finally the out tensor has a size of ([4, 23, 17]), 4 being the number of batches, 23 being the feature width after convolution, and 17 being the number of characters plus 1. The extra character is the _, used as a blank character when finding the right combination of characters for a word. When adapting values in the convolutional layers of the model, this site came in handy to calculate values for channel, width and height of the resulting feature maps.

Listing 8: The GRU-based convolutional recurrent neural network

```
1  class CRNN(nn.Module):
2   def __init__(self):
3    super(CRNN, self).__init__()
4
5    self.num_classes = (16 + 1)
6    self.image_H = 32
7
8    self.conv1 = nn.Conv2d(1, 32, kernel_size=(3, 3))
9    self.in1 = nn.InstanceNorm2d(32)
10
11   self.conv2 = nn.Conv2d(32, 32, kernel_size=(3, 3))
12   self.in2 = nn.InstanceNorm2d(32)
13
14   self.conv3 = nn.Conv2d(32, 32, kernel_size=(3, 3), stride=2)
15   self.in3 = nn.InstanceNorm2d(32)
16
17   self.conv4 = nn.Conv2d(32, 64, kernel_size=(3, 3))
18   self.in4 = nn.InstanceNorm2d(64)
19
20   self.conv5 = nn.Conv2d(64, 64, kernel_size=(3, 3))
21   self.in5 = nn.InstanceNorm2d(64)
22
23   self.conv6 = nn.Conv2d(64, 64, kernel_size=(3, 3), stride=2)
24   self.in6 = nn.InstanceNorm2d(64)
25
26   self.postconv_height = 4
```

```
  self.postconv_width = 23

  self.gru_input_size = self.postconv_height * 64
  self.gru_hidden_size = 128
  self.gru_num_layers = 2
  self.gru_h = None


  self.gru = nn.GRU(self.gru_input_size, self.gru_hidden_size, self.gru_num_layers,
    ↪ batch_first=True, bidirectional=True)

  self.fc = nn.Linear(self.gru_hidden_size * 2, self.num_classes)

 def forward(self, x):
  batch_size = x.shape[0]

  out = self.conv1(x)
  out = F.leaky_relu(out)
  out = self.in1(out)

  out = self.conv2(out)
  out = F.leaky_relu(out)
  out = self.in2(out)

  out = self.conv3(out)
  out = F.leaky_relu(out)
  out = self.in3(out)

  out = self.conv4(out)
  out = F.leaky_relu(out)
  out = self.in4(out)

  out = self.conv5(out)
  out = F.leaky_relu(out)
  out = self.in5(out)

  out = self.conv6(out)
  out = F.leaky_relu(out)
  out = self.in6(out)

  out = out.permute(0, 3, 2, 1)
  out = out.reshape(batch_size, -1, self.gru_input_size)

  out, gru_h = self.gru(out, self.gru_h)
  self.gru_h = gru_h.detach()
  out = torch.stack([F.log_softmax(self.fc(out[i])) for i in range(out.shape[0])])

  return out

 def reset_hidden(self, batch_size):
  h = torch.zeros(self.gru_num_layers * 2, batch_size, self.gru_hidden_size)
  self.gru_h = Variable(h)
```

2.4.1 LSTM-model

The model as in Listing 8 but implemented using a LSTM recurrent network can be seen in Listing 9. This model can just as well be used for the HTR-task, but on first sight it doesn't perform as well as the GRU-based model.

Listing 9: The convolutional recurrent neural network, using LSTM

```
class CRNN_lstm(nn.Module):
```

```
def __init__(self):
 super(CRNN_lstm, self).__init__()

 self.num_classes = (16 + 1)
 self.image_H = 32

 self.conv1 = nn.Conv2d(1, 32, kernel_size=(3, 3))
 self.in1 = nn.InstanceNorm2d(32)

 self.conv2 = nn.Conv2d(32, 32, kernel_size=(3, 3))
 self.in2 = nn.InstanceNorm2d(32)

 self.conv3 = nn.Conv2d(32, 32, kernel_size=(3, 3), stride=2)
 self.in3 = nn.InstanceNorm2d(32)

 self.conv4 = nn.Conv2d(32, 64, kernel_size=(3, 3))
 self.in4 = nn.InstanceNorm2d(64)

 self.conv5 = nn.Conv2d(64, 64, kernel_size=(3, 3))
 self.in5 = nn.InstanceNorm2d(64)

 self.conv6 = nn.Conv2d(64, 64, kernel_size=(3, 3), stride=2)
 self.in6 = nn.InstanceNorm2d(64)

 self.postconv_height = 4
 self.postconv_width = 23

 self.lstm_input_size = self.postconv_height * 64
 self.lstm_hidden_size = 128
 self.lstm_num_layers = 2
 self.lstm_h = None
 self.lstm_c = None

 self.lstm = nn.LSTM(self.lstm_input_size, self.lstm_hidden_size, self.
   ↪ lstm_num_layers, batch_first=True, bidirectional=True)

 self.fc = nn.Linear(self.lstm_hidden_size * 2, self.num_classes)

def forward(self, x):
 batch_size = x.shape[0]

 out = self.conv1(x)
 out = F.leaky_relu(out)
 out = self.in1(out)

 out = self.conv2(out)
 out = F.leaky_relu(out)
 out = self.in2(out)

 out = self.conv3(out)
 out = F.leaky_relu(out)
 out = self.in3(out)

 out = self.conv4(out)
 out = F.leaky_relu(out)
 out = self.in4(out)

 out = self.conv5(out)
 out = F.leaky_relu(out)
 out = self.in5(out)

 out = self.conv6(out)
```

```
  out = F.leaky_relu(out)
  out = self.in6(out)

  out = out.permute(0, 3, 2, 1)
  out = out.reshape(batch_size, -1, self.lstm_input_size)

  out, (lstm_h, lstm_c) = self.lstm(out, (self.lstm_h, self.lstm_c))
  self.lstm_h = lstm_h.detach()
  self.lstm_c = lstm_c.detach()
  out = torch.stack([F.log_softmax(self.fc(out[i]), 1) for i in range(out.shape[0])
    ↪ ])
  return out

def reset_hidden(self, batch_size):
  h = torch.zeros(self.lstm_num_layers * 2, batch_size, self.lstm_hidden_size)
  c = torch.zeros(self.lstm_num_layers * 2, batch_size, self.lstm_hidden_size)
  self.lstm_h = Variable(h)
  self.lstm_c = Variable(c)
```

2.5 RNNs, LSTMs and GRUs

To add some ideas on RNNs, LSTMs and GRUs, I consulted [1].

Basic RNNs A recurrent neural network (RNN) is an extension of a conventional feedforward neural network, which is able to handle a variable-length sequence input. The RNN handles the variable-length sequence by having a recurrent hidden state whose activation at each time is dependent on that of the previous time.

Unfortunately, it has been observed that it is difficult to train RNNs to capture long-term dependencies because the gradients tend to either vanish (most of the time) or explode (rarely, but with severe effects).

There have been two dominant approaches by which many researchers have tried to reduce the negative impacts of this issue. One such approach is to devise a better learning algorithm than a simple **stochastic gradient descent**, for example using the very simple **clipped gradient**. The other approach, is to design a more sophisticated activation function than a usual activation function, consisting of affine transformation followed by a simple element-wise nonlinearity by using gating units.

2.5.1 LSTMs and GRUs

The earliest attempt in this direction resulted in an activation function, or a recurrent unit, called a **long short-term memory (LSTM) unit**. More recently, another type of recurrent unit, **a gated recurrent unit (GRU)**, was proposed. RNNs employing either of these recurrent units have been shown to perform well in tasks that require capturing long-term dependencies.

The most prominent feature shared between GRUs and LSTMs is the additive component of their update from t to t + 1, which is lacking in the traditional recurrent unit. The traditional recurrent unit always replaces the activation, or the content of a unit with a new value computed from the current input and the previous hidden state. On the other hand, both LSTM unit and GRU keep the existing content and add the new content on top of it.

Memory This additive nature has two advantages. First, it is easy for each unit to remember the existence of a specific feature in the input stream for a long series of steps. Any important feature, decided by either the forget gate of the LSTMs unit or the update gate of the GRU, will not be overwritten but be maintained as it is.

Shortcut paths Second, and perhaps more importantly, this addition effectively creates shortcut paths that bypass multiple temporal steps. These shortcuts allow the error to be back-propagated easily without too quickly vanishing (if the gating unit is nearly saturated at 1) as a result of passing through multiple, bounded nonlinearities, thus reducing the difficulty due to vanishing gradients.

Differences between LSTMs and GRUs One feature of the LSTM unit that is missing from the GRU is the controlled exposure of the memory content. In the LSTM unit, the amount of the memory content that is seen, or used by other units in the network is controlled by the output gate. On the other hand the GRU exposes its full content without any control.

The LSTM unit computes the new memory content without any separate control of the amount of information flowing from the previous time step. Rather, the LSTM unit controls the amount of the new memory content being added to the memory cell independently from the forget gate. On the other hand, the GRU controls the information flow from the previous activation when computing the new, candidate activation, but does not independently control the amount of the candidate activation being added (the control is tied via the update gate).

2.6 Training and testing

Before starting to train the model. Some intuition has to be established concerning optimizing a Deep Learning model and more specific a HTR model. I have used this site to get some intuition on CTCloss.

2.6.1 Criterion: CTCLoss

Connectionist Temporal Classification Loss (CTC) A Connectionist Temporal Classification Loss, or CTCLoss, is designed for tasks when an alignment between sequences is needed, but where finding that alignment is difficult. Difficulties come from characters being written wider (see Figure 7) than 'normal'.

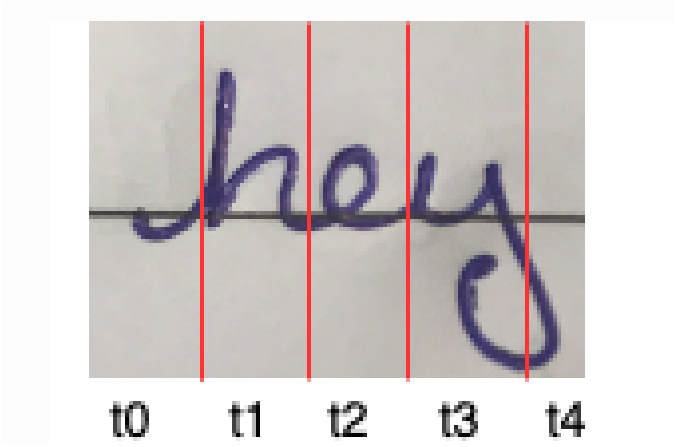

Figure 6: Image results in _hey_.
In this image every character takes up one timestep.

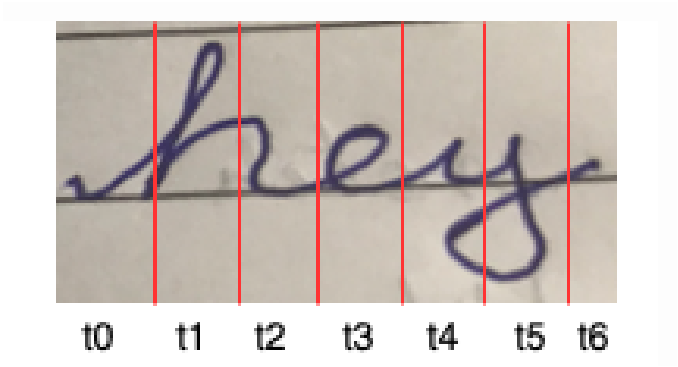

Figure 7: Image results in _hheyy_.
In this image some characters take up more than one timestep.

Collapsing characters To solve the issue, CTC merges all the repeating characters into a single character. For example, if the word in the image is 'hey' where 'h' takes three time-steps, 'e' and 'y' take one time-step each. Then the output from the network using CTC will be 'hhhey', which as per our encoding scheme, gets collapsed to 'hey'.

Pseudo-character For handling words with repeating characters, CTC uses a pseudo-character. While encoding the text, if a character repeats, then this pseudo-character is placed between the characters in the output text. Considering the word 'meet', possible encodings for it will be, 'mm_ee_ee_t', 'mmm_e_e_ttt', wrong encoding will be 'mm_eee_tt', as it'll result in 'met' when decoded.

Calculating loss To train the model, the loss needs to be calculated given the image and its label. The model as given in Listing 8 results in a probability distribution for every character possible, for every time step, see line 71. This results in a table like Table 1. Every time step the probabilities should sum up to 1.

	t0	t1	t2
a	0.4	0.3	0.4
b	0.5	0	0.0
...			
_	0.1	0.7	0.6

Table 1: Table representing some probability distribution for every time step.

Corresponding character scores are multiplied together to get the score for one path. In Table 1 the score for the path 'a__' is 0.4x0.7x0.6 = 0.168, and for the path 'aaa' is 0.4x0.3x0.4 = 0.048. For getting the score corresponding to given ground truth, scores of all the paths to the corresponding text are summed up.

For example, if the ground truth is 'a', all the possible paths for 'a' in Table 1 are 'aaa', '_a_', 'a__', 'aa_', 'a_a', '__a'. Summing up the scores of the individual paths we get, 0.048 + 0.168 + 0.018 + 0.072 + 0.012 + 0.028 = 0.346. 0.346 is the probability of the ground truth occurring, it is not the loss. To maximize this probability, all other probabilities need to be minimized.

Decoding The easiest decoding method is the greedy one, for every time step pick the character with the highest probability, and then collapse the result. The beam search searches for a combination of characters that has the highest probability.

2.6.2 Optimizer: Adam

The Adaptive Movement Estimation algorithm, or Adam is an optimization algorithm. It is an extension to stochastic gradient descent (SGD). While gradient descent bases its gradient on the complete dataset,

SGD uses only a subset of it. ADAM automatically adapts a learning rate for each input variable for the objective function and further smooths the search process by using an exponentially decreasing moving average of the gradient to make updates to variables.

2.6.3 Parameters

The training of the model is batched and is done using the loader created in Listing 3. Other parameters are in Listing 10.

Listing 10: Parameters for training

```
num_chars = 6
BLANK_LABEL = 15
crnn = CRNN().to(device)
criterion = nn.CTCLoss(blank=BLANK_LABEL, reduction='mean', zero_infinity=True)
optimizer = torch.optim.Adam(crnn.parameters(), lr=0.001)
```

2.6.4 Epochs

The train_loader contains 800 images (X) and the respective labels (label tensors containing six integers). The test_loader contains 200 such items. Train-function and test function is repeated for multiple epochs. On line 5 and 6 loss is being added to a list, see Listing 11. When all epochs have finished, loss is being saved, see line 12 and 14.

Listing 11: Training and testing for multiple epochs

```
for epoch in range(MAX_EPOCHS):
 training_loss = train(trl, crnn, optimizer, criterion, BLANK_LABEL,
    ↪ text_label_max_length)
 testing_loss = test(tl, crnn, optimizer, criterion, BLANK_LABEL,
    ↪ text_label_max_length)

 list_training_loss.append(training_loss)
 list_testing_loss.append(testing_loss)

 if epoch == 4:
  print('training loss', list_training_loss)
  print('testing loss', list_testing_loss)
  with open(generated_data_dir() + 'list_training_loss.pkl', 'wb') as f1:
   pickle.dump(list_training_loss, f1)
  with open(generated_data_dir() + 'list_testing_loss.pkl', 'wb') as f2:
   pickle.dump(list_testing_loss, f2)
  break
```

2.6.5 Training

During development I have used a dataset containing 800 unique items. A train_loader contains 80% of the dataset, being 640 items. During 5 epochs 160 batches of 4 elements are being used to train the model. So every item is used 5 times (the number of epochs) to train the model. Every item in the batch will used to test after the model is trained using the 4 items in the batch. For every mini-batch the hidden state of the RNN of GRU-units/LSTM-unit/RNN-units is being zeroed out, see line 10 of Listing 12. On line 12 a channel is added to the tensor containing the greyscaled image. For every mini-batch the optimizer's gradients are to be zeroed out, this happens on line 14. On line 17 y_pred is a tensor of size([23,4,17]). Remember from the model that the width of the feature maps was 23, 4 is the batch size, and 17 is the number of characters (including pseudo-character). On line 24 gradients are being calculated for parameters that are learnable. On line 25 parameters are updated based on the current gradients and the update rule, which depends on the optimizer. On line 27 for every image in the batch the most probable combination of characters and pseudo-characters is computed. As the feature map is 23 pixels wide, these combinations are 23 characters wide, of which the most outer pixels represent the pseudo-character.

Listing 12: Training

```
def train(train_loader, crnn, optimizer, criterion, blank_label, num_chars):
 correct = 0
 total = 0
 total_loss = 0
 num_batches = 0

 for batch_id, (x_train, y_train) in enumerate(train_loader):

  batch_size = x_train.shape[0]
  crnn.reset_hidden(batch_size)

  x_train = x_train.view(x_train.shape[0], 1, x_train.shape[1], x_train.shape[2])

  optimizer.zero_grad()

  y_pred = crnn(x_train)
  y_pred = y_pred.permute(1, 0, 2)

  input_lengths = torch.IntTensor(batch_size).fill_(crnn.postconv_width)
  target_lengths = torch.IntTensor([len(t) for t in y_train])
  loss = criterion(y_pred, y_train, input_lengths, target_lengths)
  total_loss += loss.detach().numpy()

  loss.backward()
  optimizer.step()

  _, max_index = torch.max(y_pred, dim=2)

  for i in range(batch_size):
   raw_prediction = list(max_index[:, i].numpy())

   prediction = torch.IntTensor([c for c, _ in groupby(raw_prediction) if c !=
    ↪ blank_label])
   sz = len(prediction)
   for x in range(num_chars-sz):
    prediction = torch.cat((prediction, torch.IntTensor([16])),0)


    if len(prediction) == len(y_train[i]) and torch.all(prediction.eq(y_train[i])):
     correct += 1
    total += 1

 num_batches += 1


ratio = correct / total
print('TRAIN correct: ', correct, '/', total, ' P:', ratio)

return total_loss / num_batches
```

raw prediction A raw prediction (line 30 on Listing 12 for one item in a batch looks like:

raw_prediction: [15, 15, 15, 15, 15, 15, 14, 13, 13, 16, 16, 16, 16, 16, 16, 16, 16, 16, 15, 15, 15, 15, 15]

15 is the blank_label, a pseudo-character 16 is an empty string
14 is o
13 is n

This raw_prediction is released from the pseudo-characters it contains on line 32. And if the resulting length is smaller than the maximum length (6), the absent characters are repeatedly filled with the number 16, representing an empty string. This way the prediction can be compared with the label, see line 38.

2.6.6 Testing

The testing is purely used for testing. The loader contains 160 items, and a batch contains 1 item. Testing is also repeated for 5 epochs. When compared to the training code in Listing 12, the code here contains no training code, apart from that, it is very similar to the training code. Except, in this part the character error rate (CER) and word error rate (WER) are computed, respectively on line 47 and 46. To compute the CER a function IntToString is used to convert a string from a tensor. To compute the CER a list of labels and a list of hypotheses is needed (line 33 and 34).

Listing 13: Testing

```
1  def test(int_to_char_map, loader, crnn, optimizer, criterion, blank_label, num_chars
       ↪ ):
2   int_to_string = IntToString(int_to_char_map)
3   list_of_words = list()
4   list_of_hypotheses = list()
5   correct = 0
6   total = 0
7   num_batches = 0
8   total_loss = 0
9
10  for batch_id, (x_test, y_test) in enumerate(loader):
11   batch_size = x_test.shape[0]
12   crnn.reset_hidden(batch_size)
13
14   x_test = x_test.view(x_test.shape[0], 1, x_test.shape[1], x_test.shape[2])
15
16   y_pred = crnn(x_test)
17   y_pred = y_pred.permute(1, 0, 2)
18
19   input_lengths = torch.IntTensor(batch_size).fill_(crnn.postconv_width)
20   target_lengths = torch.IntTensor([len(t) for t in y_test])
21
22   loss = criterion(y_pred, y_test, input_lengths, target_lengths)
23
24   total_loss += loss.detach().numpy()
25
26   _, max_index = torch.max(y_pred, dim=2)
27
28   for i in range(batch_size):
29    raw_prediction = list(max_index[:, i].numpy())
30    prediction = torch.IntTensor([c for c, _ in groupby(raw_prediction) if c !=
       ↪ blank_label])
31    prediction_as_string= int_to_string(prediction)
32    y_test_as_string =  int_to_string(y_test[i])
33    list_of_hypotheses.append(prediction_as_string)
34    list_of_words.append(y_test_as_string)
35    sz = len(prediction)
36    for x in range(num_chars - sz):
37     prediction = torch.cat((prediction, torch.IntTensor([16])), 0)
38
39    if len(prediction) == len(y_test[i]) and torch.all(prediction.eq(y_test[i])):
40     correct += 1
41
42    total += 1
43   num_batches += 1
```

```
 ratio = correct / total
 wer = (total - correct) / total
 cer = jiwer.cer(list_of_words, list_of_hypotheses)
 print('wer:', wer)
 print('cer:', cer)
 print('TEST correct: ', correct, '/', total, ' P:', ratio)

 return total_loss / num_batches , wer, cer
```

InToString To convert the tensor to a string the class defined in Listing 14 is used.

Listing 14: Convert IntTensor to string

```
class IntToString:
 def __init__(self, int_to_char_map):
  self.int_map = int_to_char_map

 def __call__(self, integer_tensor):
  char_sequence = []
  for i in integer_tensor:
   if i == ' ':
    ch = self.int_map['']
   else:
    ch = self.int_map[str(i.item())]
   char_sequence.append(ch)
  string = "".join([str(c) for c in char_sequence])
  return string
```

2.6.7 Visualizing loss

When training and testing the model, some values like the testing and training loss per epoch were put into lists and serialized to pickle files. These objects are restored and used to visualize loss for training and testing.

Listing 15: Visualize loss

```
with open(generated_data_dir() + 'list_training_loss.pkl', 'rb') as f1:
 list_training_loss = pickle.load(f1)
with open(generated_data_dir() + 'list_testing_loss.pkl', 'rb') as f2:
 list_testing_loss = pickle.load(f2)
epochs = range(1, len(list_training_loss)+1)
plt.plot(epochs, list_training_loss, 'g', label='Training loss')
plt.plot(epochs, list_testing_loss, 'b', label='Testing loss')
plt.xticks(range(1, len(list_training_loss)+1))
plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

The code in Listing 15 creates a figure like in Figure 8.

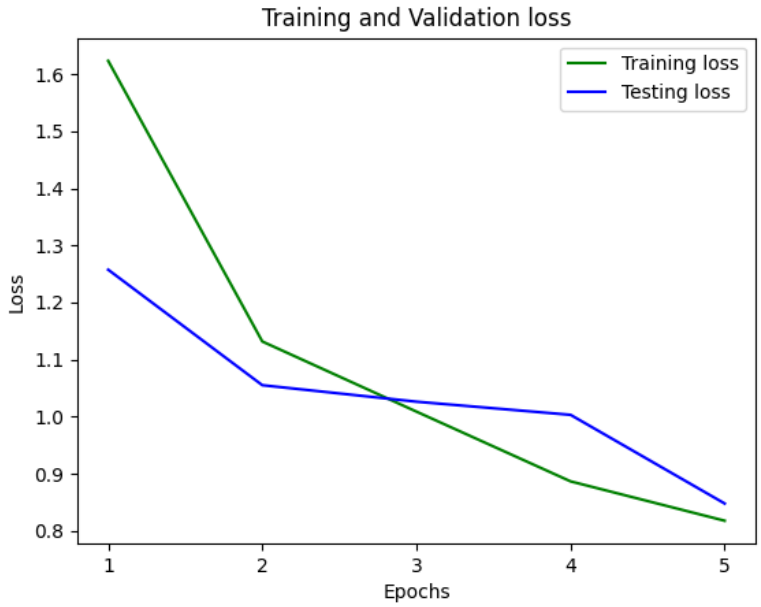

Figure 8: Loss for test en train epochs

2.6.8 Comparing error rates for different models

From Figure 9 can be seen that the model that uses a recurrent layer composed of GRU-units has the lowest character and word error rate. The LSTM-model performs worse and the basic RNN-model performs even worser. It should be remarked that models were trained using a dataloader containing 4000 unique items, of which 3200 items were used for training and 800 for testing. There were 5 epochs, during every epoch the model was trained 800 times 4 (batch-size), so every epoch trained the model using 3200 items. After every epoch training session, 800 images were tested against the model.

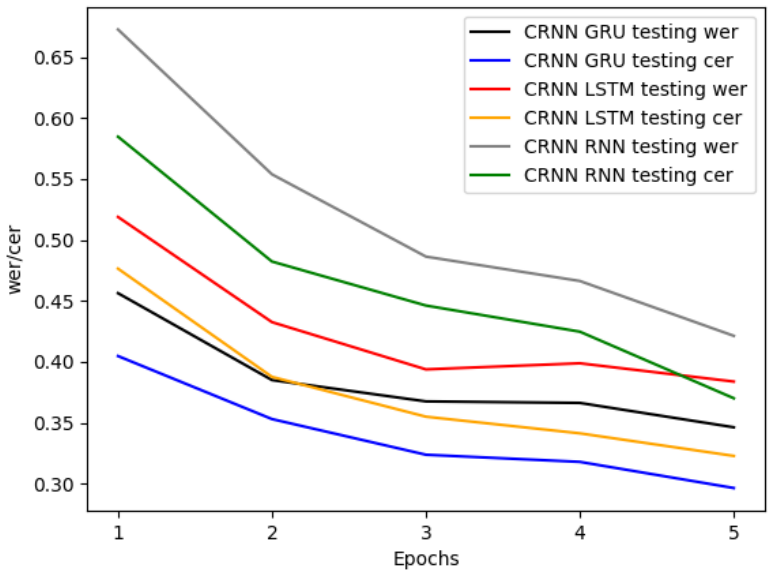

Figure 9: Comparison of the rnn, gru and lstm model

Listing 16: Visualize error rates for models

```
with open(generated_data_dir() + 'gru_list_testing_wer.pkl', 'rb') as f3:
 gru_list_testing_wer = pickle.load(f3)
```

```
with open(generated_data_dir() + 'gru_list_testing_cer.pkl', 'rb') as f4:
 gru_list_testing_cer = pickle.load(f4)
with open(generated_data_dir() + 'lstm_list_testing_wer.pkl', 'rb') as f5:
 lstm_list_testing_wer = pickle.load(f5)
with open(generated_data_dir() + 'lstm_list_testing_cer.pkl', 'rb') as f6:
 lstm_list_testing_cer = pickle.load(f6)
with open(generated_data_dir() + 'rnn_list_testing_wer.pkl', 'rb') as f1:
 rnn_list_testing_wer = pickle.load(f1)
with open(generated_data_dir() + 'rnn_list_testing_cer.pkl', 'rb') as f2:
 rnn_list_testing_cer = pickle.load(f2)
epochs = range(1, len(lstm_list_testing_wer) + 1)
plt.plot(epochs, gru_list_testing_wer,  label='CRNN GRU testing wer',  color='black'
    ↪ )
plt.plot(epochs, gru_list_testing_cer,  label='CRNN GRU testing cer', color = 'blue'
    ↪ )
plt.plot(epochs, lstm_list_testing_wer, label='CRNN LSTM testing wer',  color='red')
plt.plot(epochs, lstm_list_testing_cer, label='CRNN LSTM testing cer', color = '
    ↪ orange')
plt.plot(epochs, rnn_list_testing_wer, label='CRNN RNN testing wer',  color='grey')
plt.plot(epochs, rnn_list_testing_cer, label='CRNN RNN testing cer', color = 'green'
    ↪ )
plt.xticks(range(1, len(lstm_list_testing_wer) + 1))
plt.xlabel('Epochs')
plt.ylabel('wer/cer')
plt.legend()
plt.show()
```

2.7 Visualizing feature maps

2.7.1 Code to visualize feature maps

The code in Listing 17 shows an image and its feature map, it uses functions that are to be seen in Listing 18, Listing 19. The code that calls the visualize feature map function is in Listing 20.

Listing 17: Visualize feature maps

```
def visualize_featuremaps(crnn, loader, number):
 for batch_id, (x_test, y_test) in enumerate(loader):
  for j in range(len(x_test)):
   plt.imshow(x_test[j], cmap='gray')
   plt.show()
   img = x_test[j].unsqueeze(0)
   img = img.unsqueeze(1)
   out = crnn.simple_forward(img)
   conv_layer_plot(nrows=16, ncols=4, title='', image=out)
   number -= 1
   if number <= 0:
    break
  if number <= 0:
   break
```

2.7.2 Code to pass an image through the convolutional, activation and normalization layers

To visualize the feature maps a simple forward was added to the model, this function calculates the result of passing an already transformed image through all convolutional, activation and normalization layers.

Listing 18: Simple forward

```
def simple_forward(self, x):
```

```
 out = self.conv1(x)
 out = F.leaky_relu(out)
 out = self.in1(out)

 out = self.conv2(out)
 out = F.leaky_relu(out)
 out = self.in2(out)

 out = self.conv3(out)
 out = F.leaky_relu(out)
 out = self.in3(out)

 out = self.conv4(out)
 out = F.leaky_relu(out)
 out = self.in4(out)

 out = self.conv5(out)
 out = F.leaky_relu(out)
 out = self.in5(out)

 out = self.conv6(out)
 out = F.leaky_relu(out)
 out = self.in6(out)

 return out.permute(0, 2, 3, 1).detach().numpy()
```

2.7.3 Plotting the maps in a grid

Furthermore a function was needed to show the feature maps in a grid, see Listing 19.

Listing 19: Print maps in a grid

```
def conv_layer_plot(nrows, ncols, title, image, figsize=(14, 3), color='gray'):
 fig, axs = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)
 fig.suptitle(title)
 for i in range(nrows * ncols):
  image_plot = axs[i // ncols, i % ncols].imshow(image[0, :, :, i], cmap=color)
  axs[i // ncols, i % ncols].axis('off')
 fig.subplots_adjust(right=0.8, top=0.98, bottom=0.02, hspace=0.2)
 plt.show()
```

2.7.4 Calling the visualize feature maps function

To call the visualize feature map function, the code in Listing 20 is used. It loads a previously trained crnn, for a trained crnn will result in more meaningful feature maps.

Listing 20: Calling visualize feature maps

```
char_to_int_map, _, char_set = read_maps()
crnn = CRNN().to(device)
crnn.load_state_dict(torch.load(generated_data_dir() + 'trained_reader'))
trl, _ = get_dataloaders(image_transform, char_to_int_map, 5, text_label_max_length,
    ↪ char_set)
visualize_featuremaps(crnn, trl, 1)
```

2.7.5 A word and its feature map

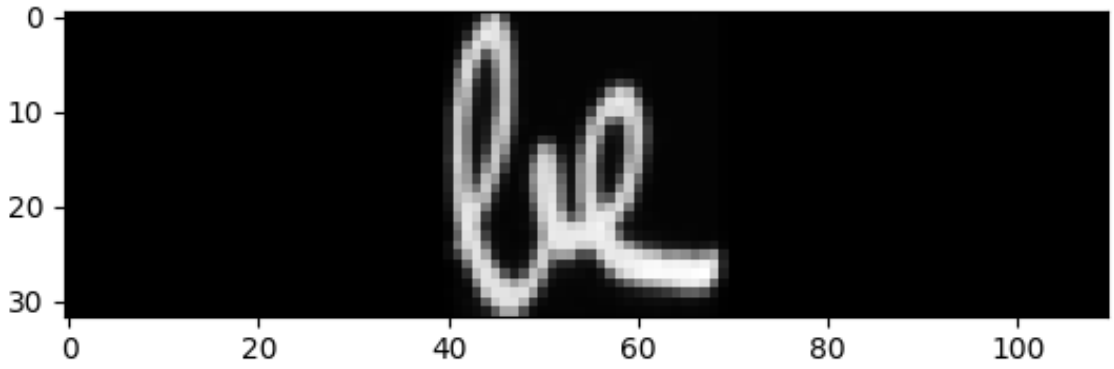

Figure 10: Image representing the word ’be’.

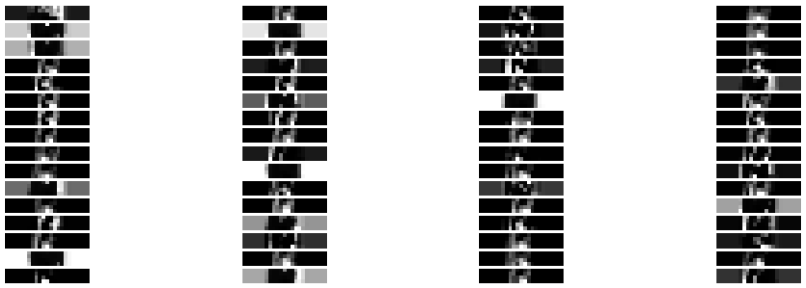

Figure 11: Feature maps for the word ’be’ in Figure 10, for every channel an image, totaling 64.

2.8 Visualizing the GRU-model

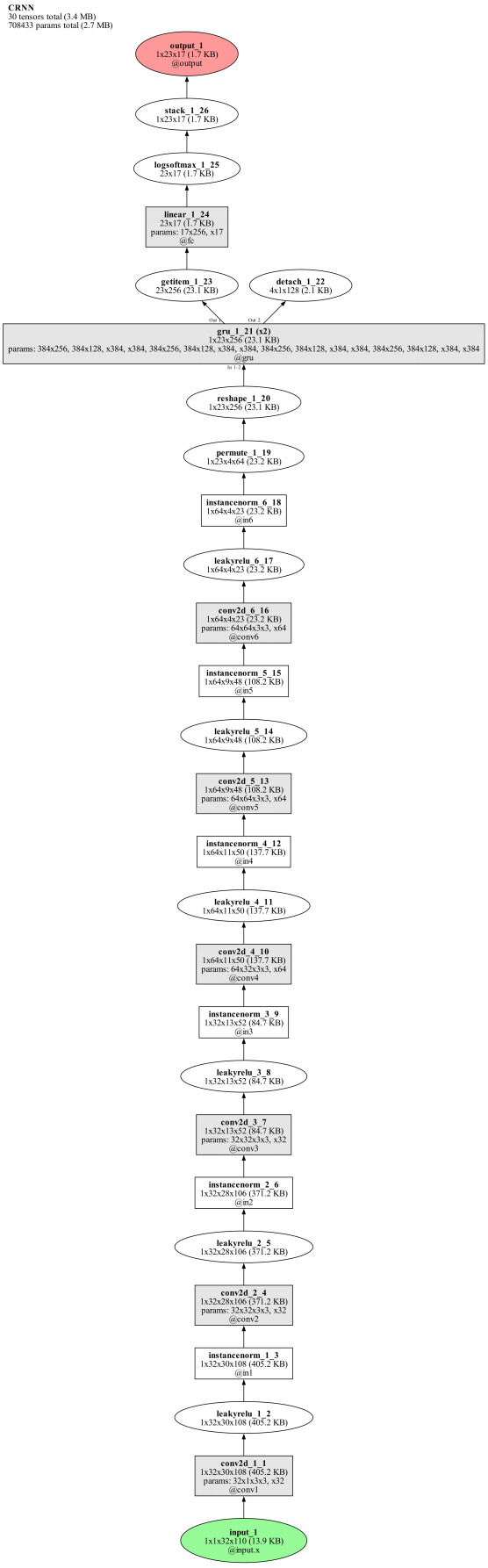

Figure 12: GRU-based model visualized by torchlens

2.8.1 Code to visualize the model

To visualize the model torchlens is used. It needs a model and an image from the dataloader. This code produces an image like in Figure 12

Listing 21: Visualizing the model

```
def visualize_model(loader, model):
 for batch_id, (x_test, y_test) in enumerate(loader):
  for j in range(len(x_test)):
   img = x_test[j].unsqueeze(0)
   img = img.unsqueeze(1)
   model_history = tl.log_forward_pass(model, img, layers_to_save='all', vis_opt='
    ↪ rolled')
   print(model_history)
```

2.9 My directories and character maps

Next to the software project directory, I have a directory called htr-torch-data, containing directories for the datasets and for the generated data. This htr-torch-data also contains a csv-file containing numbers and characters to be used for dictionaries initialization.

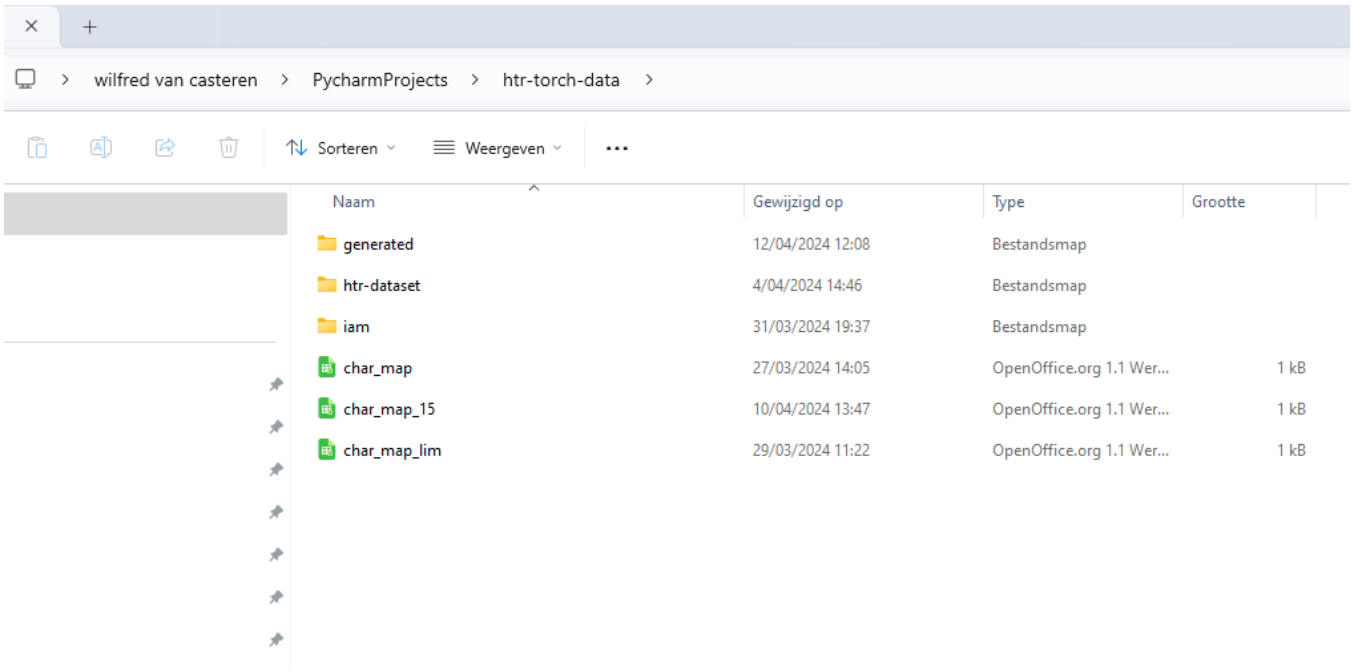

Figure 13: Directories

The code for read maps already used in several places looks like Listing 22

Listing 22: Constructing character maps and set

```
def read_maps():
 char_to_int_map = {}
 int_to_char_map = {}
 char_set = set()
 with open(external_data_dir() + 'char_map_15.csv', 'r') as file:
  csv_reader = csv.reader(file, delimiter=';', quotechar='\'', quoting=csv.
   ↪ QUOTE_MINIMAL, lineterminator='\n')
   for row in csv_reader:
    char_to_int_map[row[0]] = row[1]
```

```
    int_to_char_map[row[1]] = row[0]
    char_set.add(row[0])

 return char_to_int_map, int_to_char_map, char_set
```

3 Text object detection

To be complete I have also tried to get the d2l's single shot multibox detection working on some htr data. After a lots of hassles, I believe the problem was that the dataset used by the torch implementation of a TinySSD was trained with a dataset containing only one text object per image. To show what I wanted to achieve see Figure 14. The dataset used is to be found on this link.

3.1 Code to show bounding boxes

The code to produce the image with bounding boxes is in Listing 23

Listing 23: Visualizing bounding boxes

```
def read_bbox_csv_show_image():
 with open(htr_ds_dir() + 'train/' + '_annotations.csv', newline='') as file:
  reader = csv.reader(file)
  next(reader)
  last_image = ''
  t_bbox = torch.IntTensor()
  for row in reader:
   current_image = row[0]
   if (not current_image == last_image or last_image == '') and not len(t_bbox) ==
   ↪ 0:
    image = read_image(htr_ds_dir() + 'train/' + last_image)
    img = draw_bounding_boxes(image, t_bbox, width=5, colors=(255, 0, 0))
    img = torchvision.transforms.ToPILImage()(img)
    img.show()
    os.system('pause')
    t_bbox = torch.IntTensor()

 bbox = [int(row[4]), int(row[5]), int(row[6]), int(row[7])]
 bbox = torch.tensor(bbox, dtype=torch.int)
 bbox = bbox.unsqueeze(0)
 t_bbox = torch.cat((t_bbox, bbox), 0)
 last_image = row[0]
```

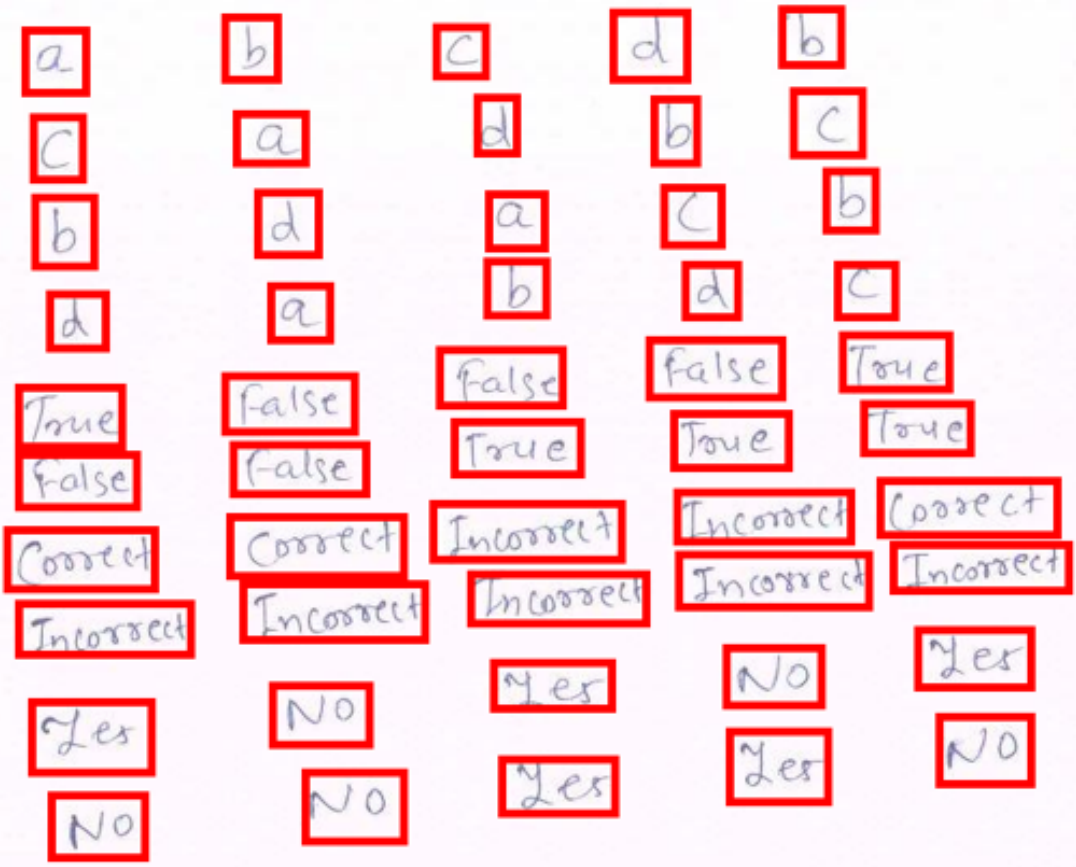

Figure 14: Text object detection example image

References

[1] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.