Research Methods for AI
A report:
Deep Learning: training and testing a bidirectional GRU-based HTR-model

Wilfred Van Casteren
nr:837377516
Open Universiteit
Masteropleiding AI

April 11, 2025

Abstract

In this document I describe some of the experiments I did concerning Handwritten Text Recognition or HTR. In the experiments I examined performance effects of applying image augmentation, a dropout layer and pre-training on a deep learning model. I present plots to visualize and compare error rates, loss and word length correctness. The most obvious conclusion I was able to draw from the experiments, was that by applying a simple form of pre-training on a HTR-model, the model became more susceptible to recognize longer words. The appendices contain some deeper insights into the technicalities of the model, its convolutional neural network (CNN), the Gated Recurrent Unit (GRU) involved and some extra plots. The model was programmed using Pytorch.

Introduction

This report contains information concerning research I conducted to investigate the performance of a handwriting recognition model, this domain is commonly referred to by the term Handwritten Text Recognition (HTR).Even though it is neither purpose nor possibility for me to reach performance levels of 98% for HTR, I would like to point you to an article, Enhancement of handwritten text recognition using AI-based hybrid approach, that, in my opinion, gives a good explanation for more complex HTR-related situations than those described in this writing.

Sequential problem 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 word(s).

Difficulties Individual characters can be connected, or can be written separately as can be seen in Figure 4, handwriting styles can differ in so many aspects. This adds to the complexity of converting images to normal strings. Furthermore images of the words themselves have all kinds of different shapes, no image is just as wide and high as any other, which makes handling them is a problem too.

Objectives

Short single words In this study I will focus on the recognition of short words (6 characters long), using a limited character set. I will not focus on complete lines, paragraphs or pages. The ANNs (Artificial Neural Network) will be trained and tested using words in a known dataset, the IAM dataset. Pre-training will be done using handwriting generated by an existing model. The dropout will be applied in the GRU-based recurrent neural network (RNN) by setting a parameter. In [3] it is stated that applying dropout should improve performance for a HTR-model.

The conducted experiments will achieve multiple things, they will:

- compare the application of three experimental conditions (dropout, image augmentations and pre-training), applied to a HTR-model, in terms of loss, error rates and word length accuracy
- put forward some of the limitations of the model presented and the experimental setup itself

Network

For the ANN a convolutional neural network (CNN) in combination with an RNN will be used. In addition a ctc loss function will be used to find the optimal character combination. The RNN used will be GRU-based (Gated Recurrent Unit). This RNN is described in a chapter on GRU.

Connectionist Temporal Classification Loss (CTCLoss)

To handle the irregularities imposed by the multitude of authors and their handwritings CTCLoss will be used. The Connectionist Temporal Classification Loss, or CTCLoss, is designed for tasks where an alignment between sequences is needed, but where finding that alignment is difficult. In [1] the CTC loss is described in more detail. A more readable explanation on ctc is on this site

CTC greedy search The most used algorithm to decode CTC is the greedy search algorithm. It selects the most probable option at each time-step. I will use this too in my experiments.

Pretraining

In the course of my research I found out my model wasn't able to 'read' words longer than 3 characters. So I used a handwriting generation model to generate more examples. I especially wanted to add longer words, I wanted these longer words to contain every character.

So for every character from (a-z), I generated words from length 1-6 containing that character, this using 8 different writing styles. As the generating model isn't perfect I had to delete some of the generated words, and could not generate as much usable examples as I wanted. Two of the generated words are in Figure 1 and Figure 2.

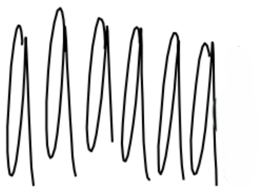


Figure 1: The word ’aaaaaa’ as generated by a handwriting model.

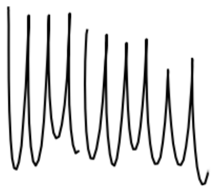


Figure 2: The word ’uuuuu’ as generated by a handwriting model.

IAM Database

The IAM database is a database which could serve as a basis for a variety of handwriting recognition tasks, it contains images of words, images of pages containing text, images of lines containing text. Some ’word’ examples from this database are shown in Figure 3 and Figure 4. It should be noted however, I will be focusing on lowercase characters, and will limit the characters involved to words containing characters a-q.

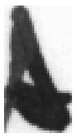


Figure 3: A word from the IAM dataset.

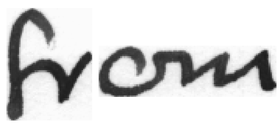


Figure 4: Another word from the IAM dataset.

Image transformations

Default image transformations

All word images will be inverted, padded and resized to the same size and converted to grayscale images. It is easier to work with equally sized images when working with Tensors. Inverting images will result in light characters on a dark background. The model that will be evaluated, will be trained and tested using grayscaled images, as images are grayscaled their shape will have one channel where normal colored images have three, so handling them will be cheaper in terms of memory and computational requirements.

Extra transformation applied when pre-training As the generated images used for pre-training seem to have been written using a thinner pencil some extra transformation is applied to thicken the lines and to add some more contrast.

Non-default image transformations

The non-default transformations are transformations that could improve the performance of the model being evaluated. They are applied when training and pre-training the model. In Listing 1 the transformation scheme is defined. The p parameter used on lines 4,7,8,9,10,11 and 13 sets the possibility the transformation will be applied. OneOf picks one transformation of possibly more transformations. Using this scheme, it is possible two different transformations will be applied on one word image, but no more.

Idea The main idea of these transformations would be to raise generalization for the trained model. The transformations could possibly emphasize deviations already present or add deviations to the word images, thereby raising awareness of the model when training on them.

No other research I haven't really found any existing research, focusing on the effects of these kind of transformations on the performance of HTR-models. Best practices for a handwritten text recognition system is a study where image augmentation is mentioned though.

Listing 1: Albumentation transformations in code

```
1  def train_transform():
2   return A.Compose(
3    [
4     A.Rotate(limit=(-4.5, 4.5), p=0.25, border_mode=cv2.BORDER_CONSTANT),
5     A.OneOf(
6      [
7       A.GaussNoise(p=1, var_limit=(233, 255), per_channel=False),
8       A.Blur(p=1, blur_limit=3),
9       A.Morphological(p=1, scale=(3, 4), operation="dilation"),
10      A.Morphological(p=1, scale=(3, 4), operation="erosion", ),
11      A.PixelDropout(p=1, drop_value=133, dropout_prob=.04)
12     ],
13     p=0.45,
14    )
15   ]
16  )
```

Example of a word transformation

By applying the previously defined transformation scheme, Albumentations will possibly add transformations, to an original image shown in Figure 5, which could result in Figure 6. The image (the word) is rotated slightly and some gaussian noise has been added. To see a live example where gaussian noise is applied, check out this link. As handwritten text already contains several natural 'transformations', the artificial transformations will be added very moderately.

When I started my research, the Albumentations library was the most flexible, but in the meantime pytorch transformations might have become just as flexible as Albumentations.

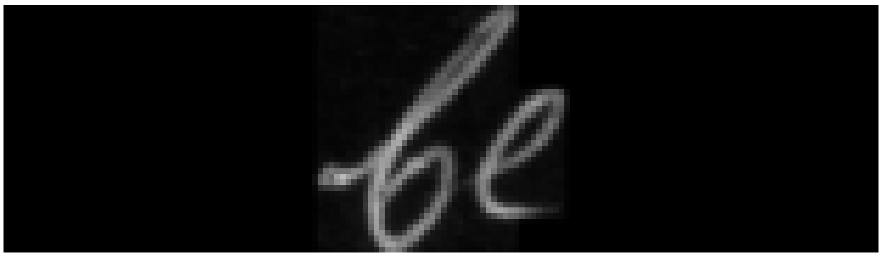


Figure 5: Untransformed word.

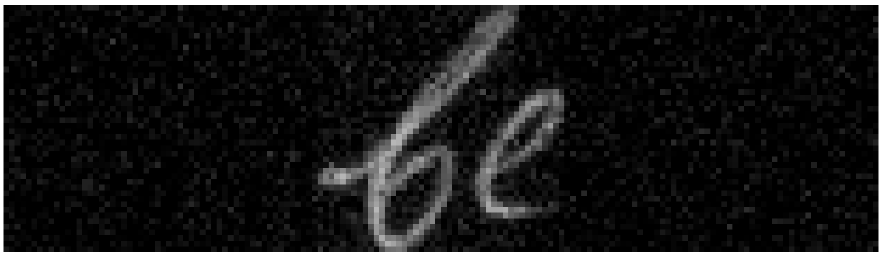


Figure 6: Transformed word.

Experimental setup

In the previous section the some important elements of my research and some of its testing conditions were described. In this section, the model as described in The model will be tested under 4 different conditions. It will be evaluated on word error rate and character error rate. I will also investigate word length correctness. The overall performance rate will be shown too. The full code to perform experiments, generate plots, check out applied transformations is in this repository.

Research Objectives (ROs)

- The model as described in the section on The model will be evaluated using original data
- The model as described in the section on The model will be evaluated using augmented data
- The model as described in the section on The model, trained using dropout (50%) in the RNN-layer, will be evaluated using original data
- The model as described in a section on The model, trained using dropout (50%) in the RNN-layer, will be evaluated using augmented data

In the most ideal world applying image augmentation and dropout should enhance performance.

Modalities

Pretraining

In the course of the research project, I came to the conclusion my model almost never made any correct guesses for words longer than 3 characters. Therefore I decided to repeat the ROs for a model pretrained using 'words' generated by a handwriting generation model. Using this model I generated words 'a' to 'zzzzzz' using 8 different handwriting styles. I used these words repeatedly to pre-train the model.

As the generated words contained quite some 'outliers', I had to go through every image to possibly delete or adapt the generated word image, but I still managed to generate a substantial amount of -be it- meaningless words.

725 unique examples The model will be pre-trained on 725 unique examples. As no other examples are available, these examples will be reused 25 times while pre-training. For every new set of conditions the same training sequence will be applied using the same set of transformations, but when in a new iteration transformations will be applied using a new seed (this seed changes by 1 every iteration). This should guarantee no example is presented to the same model transformed in the same way, though the same data is reused over and over.

Training

5 epochs, 15000 examples, 2400 training examples per epoch, 600 testing examples per epoch Evaluations will be run after every training epoch. A model is trained for 5 epochs using 2400 examples per epoch. Every epoch it is tested using 600 examples not used for training.

Equal opportunities For every set of test conditions, the dataset will be divided over the different epochs in the same way, thereby assuring every epoch uses unique data. By using the same epoch, train and evaluation splits on the same dataset for every RO, every RO is achieved using the same background conditions.

Random Albumentations augmentations, when applied, will only be applied on training data (not on testing data), on the same examples using the same random seeding. So any differences in model performance will come from changes in independent variable(s), be it the presence/absence of a dropout layer, of the image augmentation part or the pre-training part of the test conditions.

Visualizing error rates, loss and performance

While training and testing the model, some values like the loss, cer (character error rate), wer (word error rate) were put into lists and serialized to pickle files. After restoring those objects I was able to generate several plots.

Word error rate The word error rate (wer) is a quite obvious metric, if 1 out of 10 words is read wrong, the wer is 0.10.

Character error rate The character error rate (cer) is a quite obvious metric, if 1 out of 10 characters is read wrong, the cer is 0.10.

Error rates compared

Word error rates

The word error rate spread is much smaller for the model without any pre-training is an observation to be made.

No pre-training The model without any dropout in the GRU-based RNN-layer, and trained with image augmentation has the lowest word error rate. The model without dropout but trained without applying image augmentation is very close.

Pre-training The pre-trained model implementing no image augmentation, not having any dropout rate performs best.

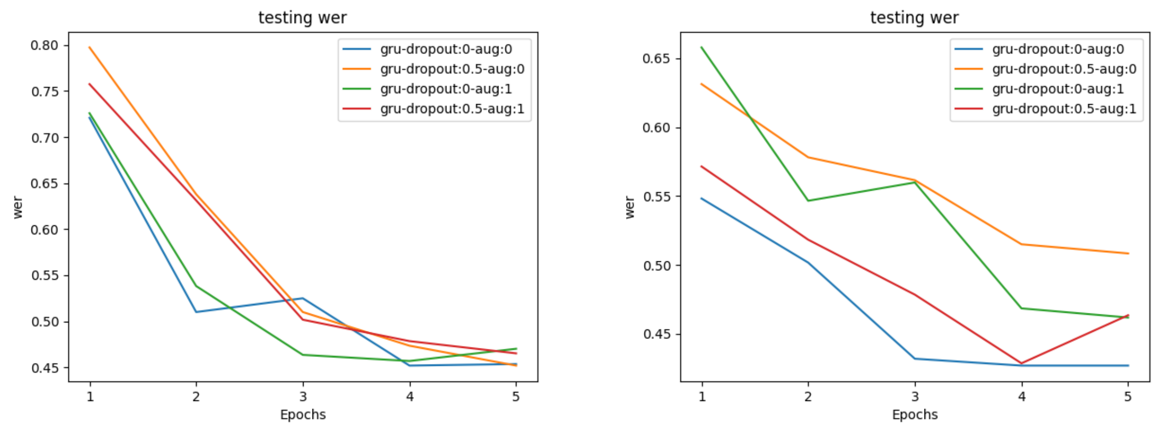


Figure 7: Word error rates using a normal model (left) and a pre-trained model (right).

Character error rates

Character error rates are generally lower for the pre-trained model.

No pre-training The model without any dropout in the GRU-based RNN-layer, and trained using the non-augmented images has the lowest character error rate.

Pre-training The pre-trained model that was trained without image augmentation not having dropout layers, achieves the lowest character error rate.

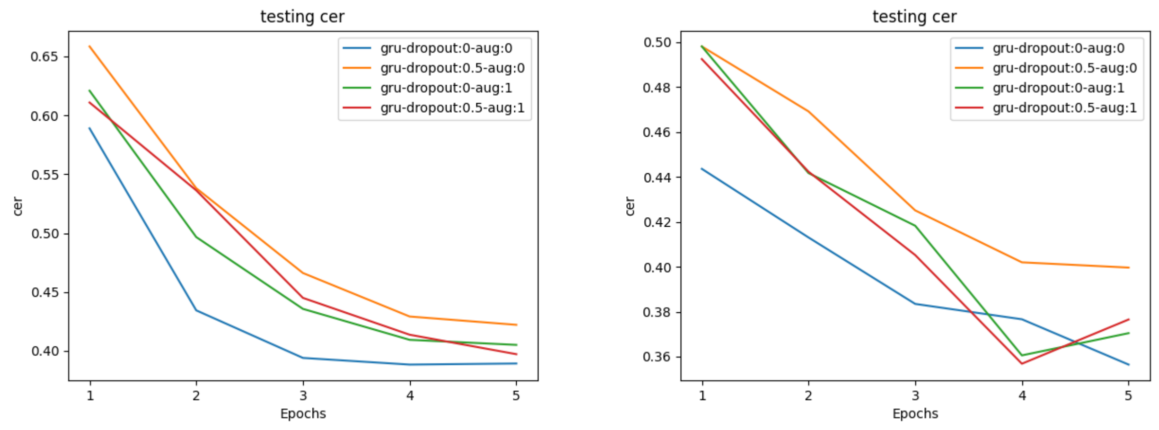


Figure 8: Character error rates using a normal model (left) and a pre-trained model (right).

Training loss

The model without any dropout in the GRU-based RNN-layer, and trained using the non-augmented images has the lowest loss for both the pre-trained or non pre-trained case.

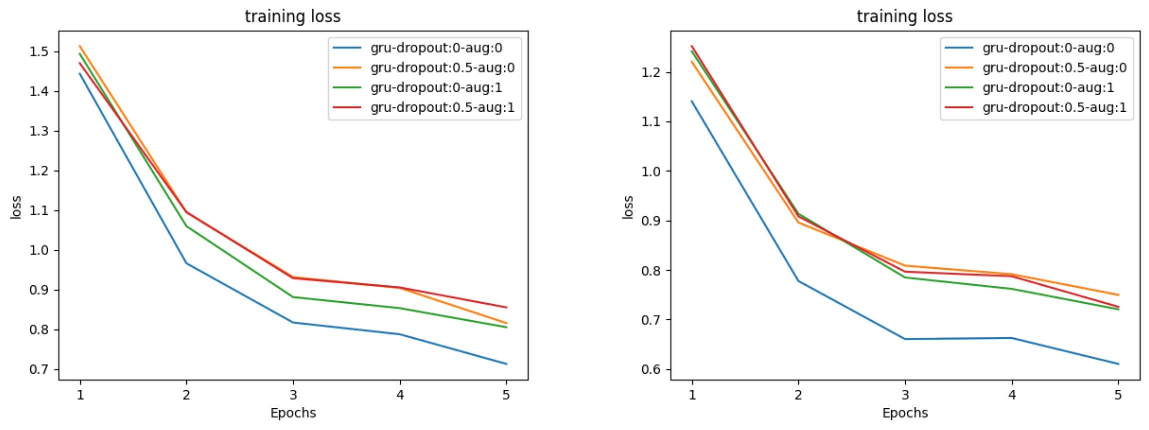


Figure 9: Loss using a normal model (left) and a pre-trained model (right).

Counts and guesses

To get a better idea of the guesses an ANN made when offered an image or the number of times a model is trained on a word I gathered data. To see what kind of data I gathered see Counts and guesses.

Word length correctness

For each of the testing conditions a comparison was made by the length of the word. When no pre-training is done, the main observation to be made,is that for every set of test conditions almost none or none of words of length 4 or more are being recognized, see Figure 10. When pre-training is done things look somewhat better when not using any dropout, see Figure 11 and Figure 12. Even some 5-letter words are being recognized.

More graphs on word length correctness are available in the appendix chapter on Word length correctness.

No dropout, no image augmentation, no pre-training

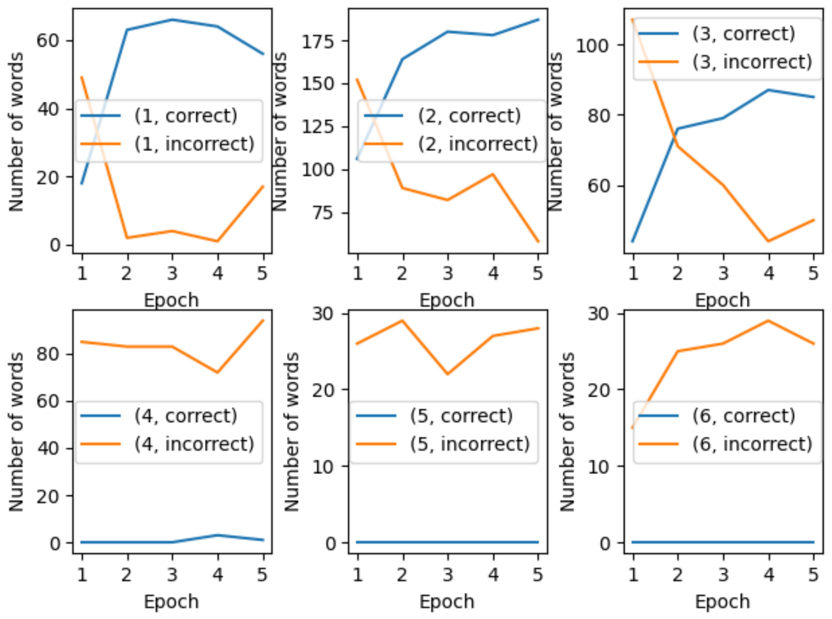


Figure 10: Words correctly recognized by length, the length of words varies from 1-6.

No dropout, no image augmentation, pre-training

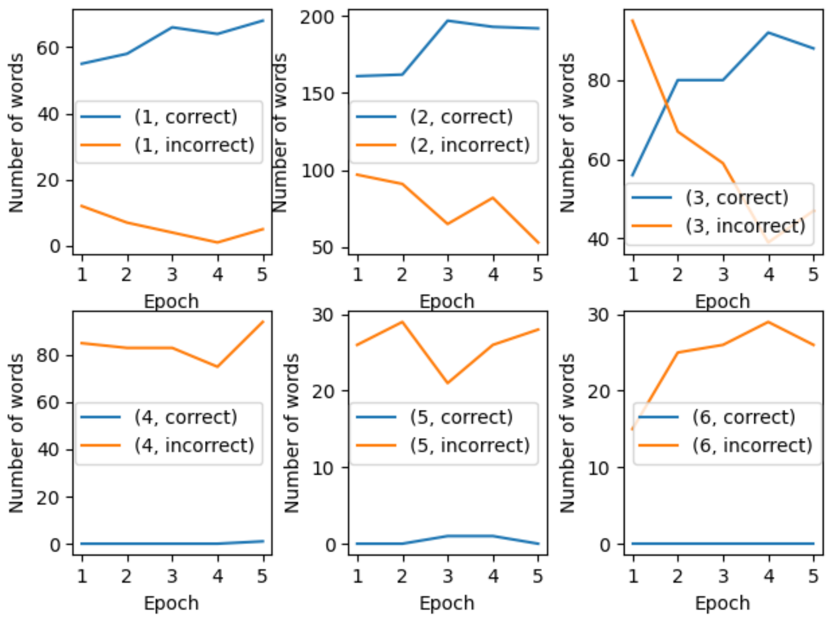


Figure 11: Words correctly recognized by length, the length of words varies from 1-6.

No dropout, image augmentation, pre-training

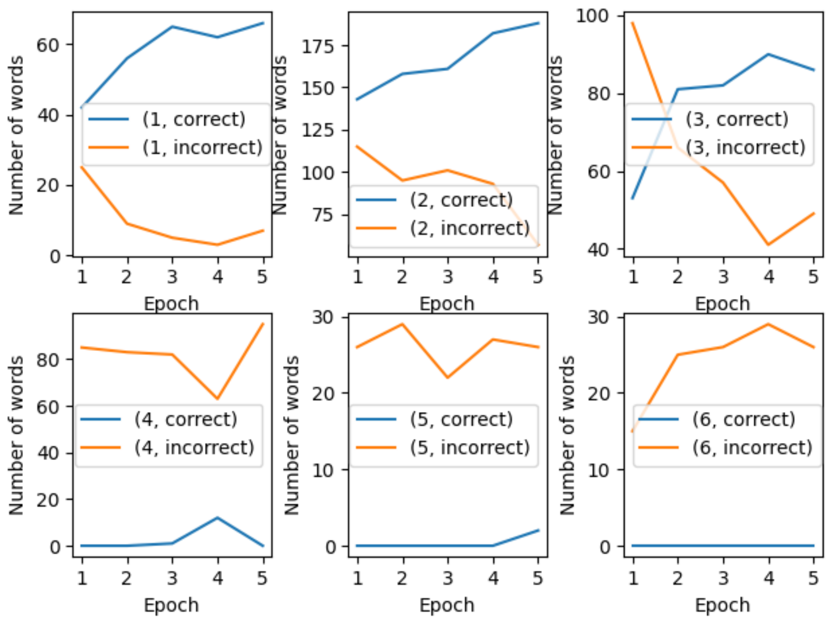
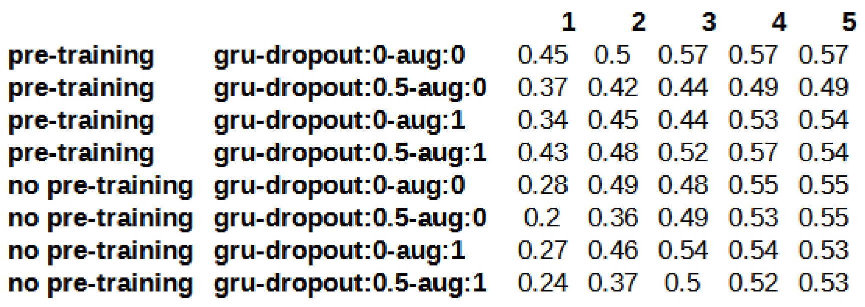


Figure 12: Words correctly recognized by length, the length of words varies from 1-6.

Overall performance

See Figure 13 for a comparison of all sets of test conditions. The best performing model is the pre-trained model without using image augmentation or dropout layer. Performance is simply based on the number of words correctly guessed.



		1	2	3	4	5
pre-training	**gru-dropout:0-aug:0**	0.45	0.5	0.57	0.57	0.57
pre-training	**gru-dropout:0.5-aug:0**	0.37	0.42	0.44	0.49	0.49
pre-training	**gru-dropout:0-aug:1**	0.34	0.45	0.44	0.53	0.54
pre-training	**gru-dropout:0.5-aug:1**	0.43	0.48	0.52	0.57	0.54
no pre-training	**gru-dropout:0-aug:0**	0.28	0.49	0.48	0.55	0.55
no pre-training	**gru-dropout:0.5-aug:0**	0.2	0.36	0.49	0.53	0.55
no pre-training	**gru-dropout:0-aug:1**	0.27	0.46	0.54	0.54	0.53
no pre-training	**gru-dropout:0.5-aug:1**	0.24	0.37	0.5	0.52	0.53

Figure 13: Performance for 8 different sets of test conditions.

Room for improvement

After this study I am **almost** convinced it is possible to reach a word recognition rate, using the described model, that is higher than 60%, 70% or even 80%, without too much effort. The best performance I was able to reach was 57% (no dropout, no augmentation, pre-trained), while the lowest was 49% (dropout, no augmentation, pre-trained). Some improvements are to be made however.

Model-specific improvements

Apply dropconnect. I have the impression, the way dropout is implemented in the GRU as conceptualized by Pytorch, differs from how [3] would apply it. Dropconnect is similar to dropout as it introduces dynamic sparsity within the model, but differs in that the sparsity is on the weights, rather than the output vectors of a layer. As the same weights are reused over multiple timesteps, the same individual dropped weights remain dropped for the entirety of the forward and backward pass, this seems a better way to retain info from earlier timesteps.

Beam Search instead of greedy search By applying Beam Search when decoding CTCLoss, some errors could disappear. Beam Search is described in [5].

Use a dictionary Of course it is always a good idea to use some kind of dictionary. A non-existing word guess could be changed to the most similar one.

Use a different CNN By using a different CNN, some writing details could probably be captured better. By changing normalization, max-pooling, widening the CNN, more details could probably be captured faster.

Add more layers to the RNN, use a wider hidden state By using a more complex RNN, some writing details could be captured/kept better.

Use other optimizer hyper-parameters Changing the hyper-parameters on the optimizer, or change the optimizer altogether could change performance.

Other improvements

Generate more (varying) examples As the model I used for generating handwritten examples created too much unreadable/incorrect words[1], I would have to find/train a model that does better[2]. Generating more varying words could prove successful in reducing outliers/unreadable words too. When going through the generated examples, I found out handwritten characters resembling the typewriter character, for example the letter 'b' looks like 'b', where I would write it like . This 'problem' needs some attention.

Character-level transformations A better way to transform word images would be to do character-level transformations instead of word-level transformations. Applying transformations on individual characters would make it easier to change the handwriting style for a given word.

Gather info on character-level alignments for model guesses As the IAM database doesn't contain info on the position of the characters that make up a word it could be interesting to find out how the ctcloss function aligns pixels to characters for correct and false guesses. After having achieved enough correct guesses for words of every length these pixel-wise alignments could be used for applying character-level image transformations. This alignment info could also be used for generating character regions on word images to facilitate interpreting errors.

[1]It might have had something to do with repeating the same characters over and over

[2]I would have tested/used other models but wasn't able to get most of the models, I found on the internet, running.

Bibliography

[1] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 369–376, New York, NY, USA, 2006. Association for Computing Machinery.

[2] Supriya Mahadevkar, Shruti Patil, and Ketan Kotecha. Enhancement of handwritten text recognition using ai-based hybrid approach. *MethodsX*, 12:102654, 2024.

[3] Vu Pham, Théodore Bluche, Christopher Kermorvant, and Jérôme Louradour. Dropout improves recurrent neural networks for handwriting recognition. In *2014 14th international conference on frontiers in handwriting recognition*, pages 285–290. IEEE, 2014.

[4] George Retsinas, Giorgos Sfikas, Basilis Gatos, and Christophoros Nikou. Best practices for a handwritten text recognition system. In *International Workshop on Document Analysis Systems*, pages 247–259. Springer, 2022.

[5] Harald Scheidl, Stefan Fiel, and Robert Sablatnig. Word Beam Search: A Connectionist Temporal Classification Decoding Algorithm . In *2018 16th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 253–258, Los Alamitos, CA, USA, August 2018. IEEE Computer Society.

Word length correctness

No pretraining

No dropout, no image augmentation

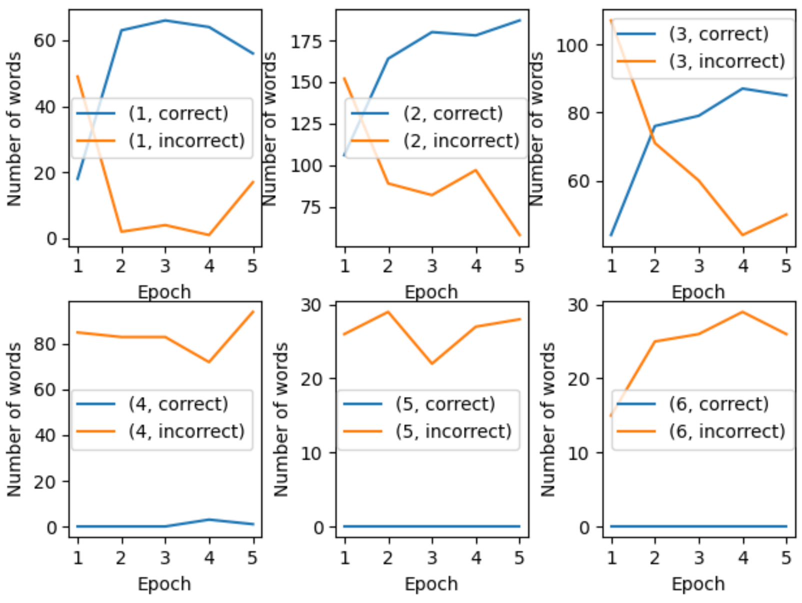


Figure 14: Words correctly recognized by length, the length of words varies from 1-6.

Dropout, no image augmentation

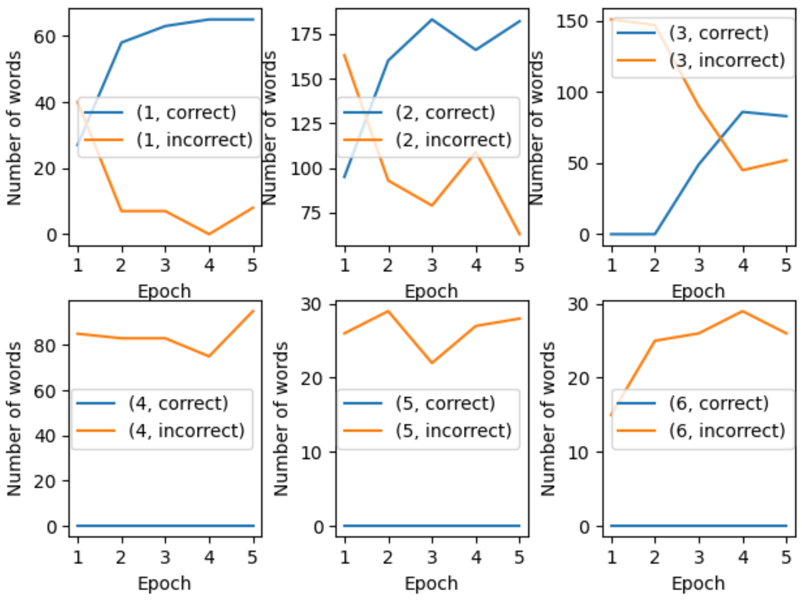


Figure 15: Words correctly recognized by length, the length of words varies from 1-6.

No dropout, image augmentation

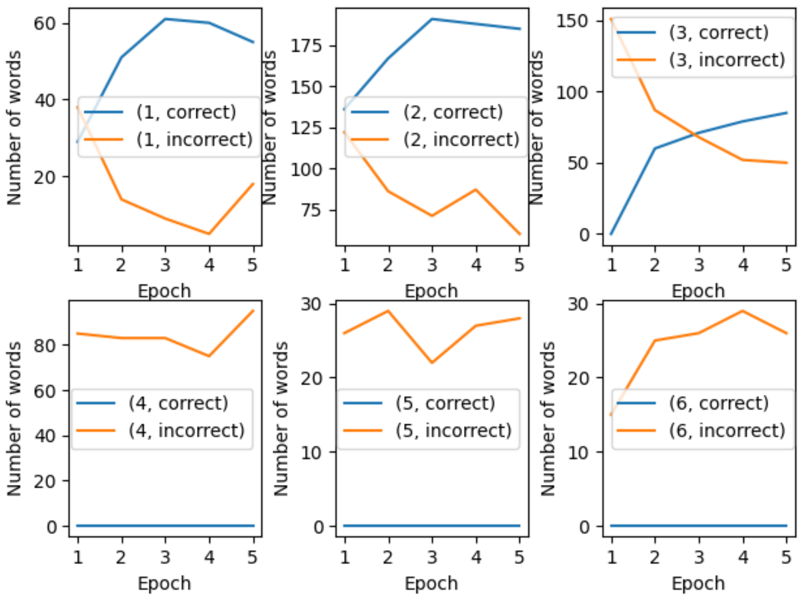


Figure 16: Words correctly recognized by length, the length of words varies from 1-6.

Dropout, image augmentation

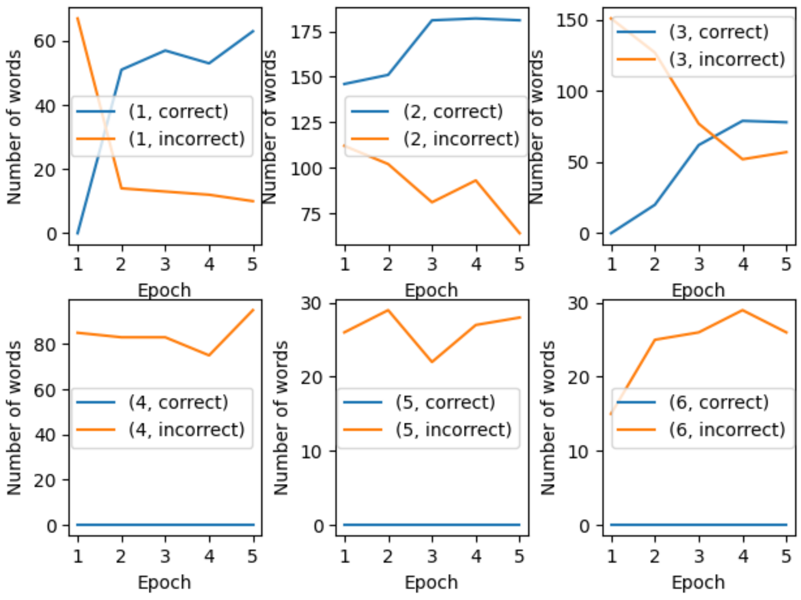


Figure 17: Words correctly recognized by length, the length of words varies from 1-6.

Pretraining

No dropout, no image augmentation

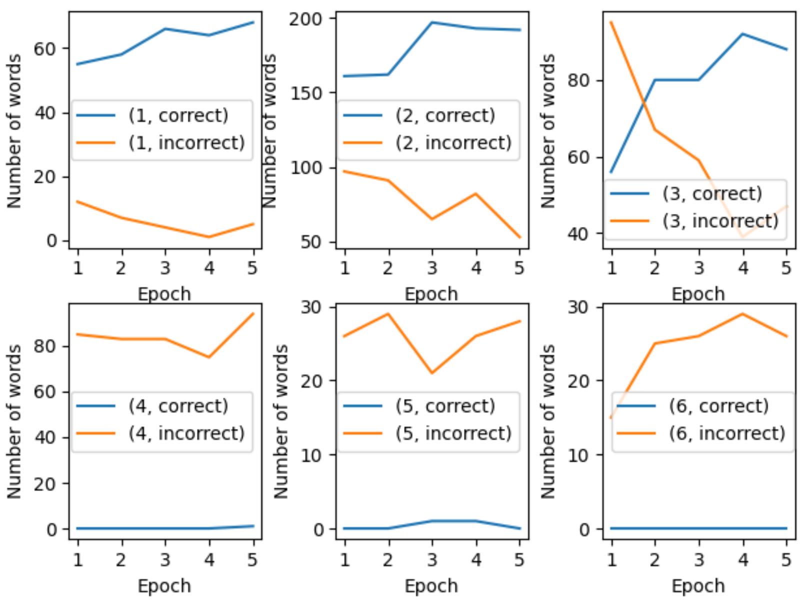


Figure 18: Words correctly recognized by length, the length of words varies from 1-6.

Dropout, no image augmentation

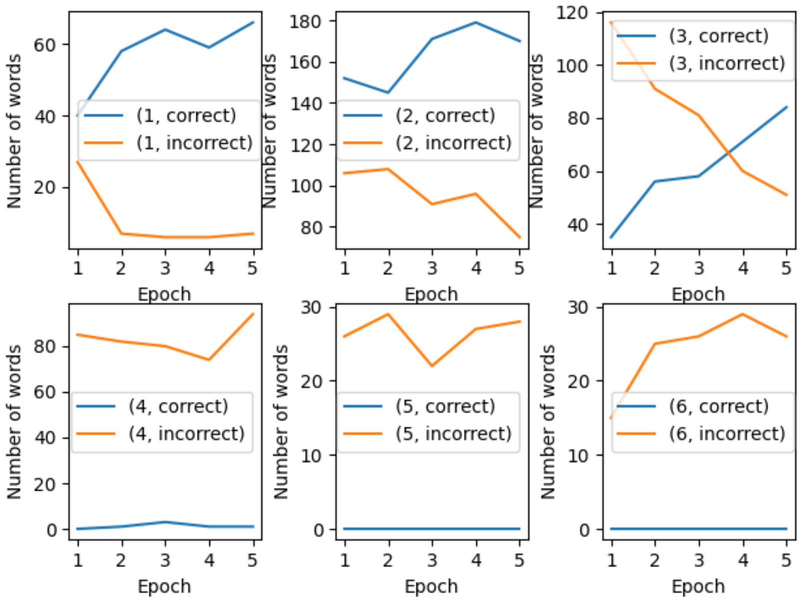


Figure 19: Words correctly recognized by length, the length of words varies from 1-6.

No dropout, image augmentation

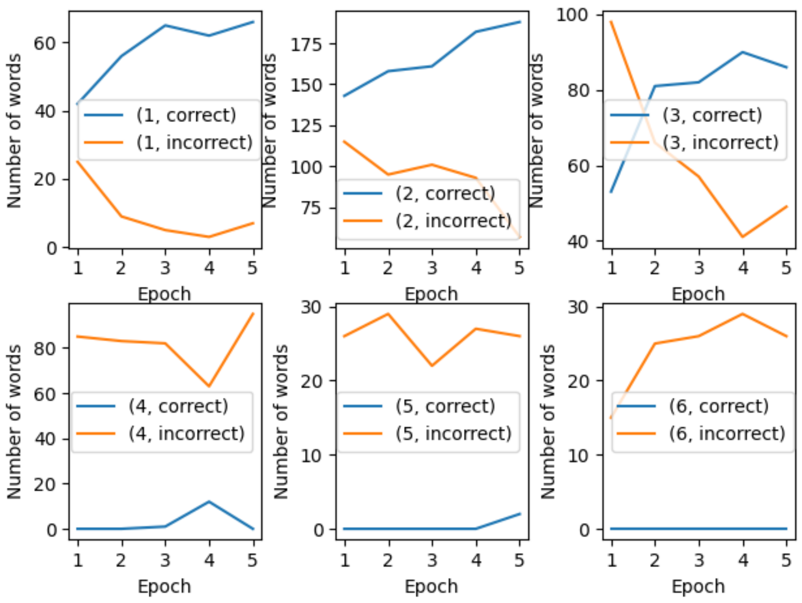


Figure 20: Words correctly recognized by length, the length of words varies from 1-6.

Dropout, image augmentation

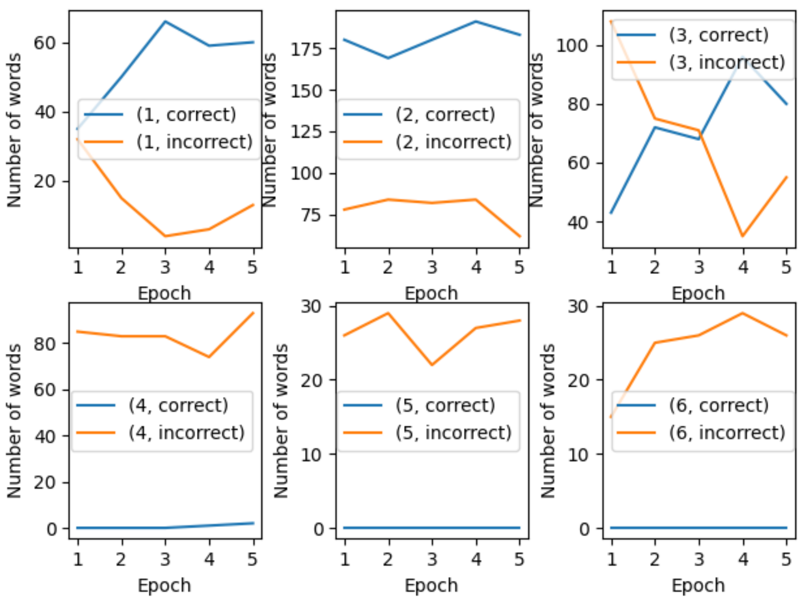


Figure 21: Words correctly recognized by length, the length of words varies from 1-6.

Counts and guesses

Counts

To get a better idea of the contents of my dataset, I created a file that keeps count of the words used when training the model, see Listing 2 for a partial view on this data.

Listing 2: Counts

```
foe,1
god,1
fan,2
nal,1
pin,1
ion,1
oil,1
deep,4
been,241
like,69
once,22
come,33
back,51
mind,25
plan,13
look,15
...
```

Guesses

To get a better idea of the guesses made by a model I kept a list per testing epoch containing truth values (left) and guesses made (right), for a partial view on this data see Listing 3.

Listing 3: Guesses

```
gone,on
of,of
of,of
be,he
of,of
be,he
in,in
line,hee
a,a
change,mann
of,of
man,mnn
on,on
a,on
of,of
a,a
he,he
on,on
again,aganl
...
```

I haven't really used the information in this subsection, it could however prove valuable.

Model and code

In this chapter some light will be shed on the model I used. Also some details on a GRU-based RNN and a CNN will be exposed. The model that was used during experiments is not very exceptional. In fact, most recent models for HTR are combination of a CNN, an RNN and some ctcloss function.

Python packages

The main packages I have used to assemble the model and program the experiments are listed in Listing 4.

Listing 4: Packages used

```
python = "^3.12"
numpy = "^1.26.4"
matplotlib = "^3.8.3"
wakepy = "^0.7.2"
jiwer = "^3.0.4"
pillow = "^10.4.0"
scikit-learn = "^1.5.2"
torch = "^2.5.1"
torchinfo = "^1.8.0"
torchvision = "^0.20.1"
torchlens = "^0.1.24"
albumentations = "1.4.23"
```

Torch will be used to create the ANN. Numpy is the default multi-dimensional array package for Python. Matplotlib is the library for creating plots and figures. Torchvision is an auxiliary library to the torch library. Wakepy is a package to prevent a computer from going to sleep. Torchlens is a package to visualize the model. Jiwer is a package to calculate error rates and other metrics. Pillow is a library for image processing. Torchinfo is an auxiliary package to the torch library. Scikit-learn is the go to library when dealing with data analysis problems.

The model

In Listing 5 a complete model is defined. In the __init__ method the model's base properties are defined. the num_classes (line 5) is an integer representing the number of characters used, in this case 15. One is added to the num_classes for the CTCLoss character (_). On line 10 and 11 the height and width after convolution are set. This height will be used to define the gru_inputsize (line 13), gru_hidden_size is a variable that will set the size of the hidden state for the GRU-layer. The GRU-layer itself is defined on lines 19-25, its input size is set, the size of its hidden state is set, the number of layers is set, its bidirectionality is set, and a dropout value is set. On line 27 a fully connected layer is defined, it will use the output of the GRU as its input, and will generate a probability distribution over every possible character (a-q) for every pixel in its first dimension (35). The zeroth dimension of the GRU-output is the size of the mini-batch and the second dimension is the height (7). It is this out variable over which the ctcloss will calculate a word guess.

Listing 5: Complete model

```
1  class CRNN(nn.Module):
2   def __init__(self, num_classes, dropout):
3    super(CRNN, self).__init__()
4
5    self.num_classes = num_classes + 1
6    self.image_H = 44
7
```

```
 self.cnn = simple_CNN()

 self.postconv_height = 7
 self.postconv_width = 35

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

 self.gru = nn.GRU(
 self.gru_input_size,
 self.gru_hidden_size,
 self.gru_num_layers,
 batch_first=True,
 bidirectional=True,
 dropout=dropout
 )
 self.fc = nn.Linear(self.gru_hidden_size * 2, self.num_classes)

def forward(self, x):
 batch_size = x.shape[0]
 out = self.cnn(x)
 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]), 1) 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)
```

CNN

The model that will be subject of our research will be a GRU-based word-level CNN HTR-model. It uses a CNN as defined in Listing 6.

CNN? A convolutional neural network consists of an input layer, hidden layers and an output layer. The hidden layers include one or more layers that perform convolutions. Typically this includes a layer that performs a dot product of the convolution kernel with the layer's input matrix. The activation function is commonly a ReLU (Rectified Linear Unit). As the convolution kernel slides along the input matrix for the layer, the convolution operation generates a feature map, which in turn contributes to the input of the next layer. This is followed by other layers such as pooling layers and normalization layers. In the CNN at hand the pooling layer is actually replaced by using a stride when applying a convolutional layer.

Normalization Instance Normalization helps to stabilize and improve the training of neural networks by normalizing each instance (or sample) independently within a mini-batch. It calculates the mean and variance for each feature map in each instance, normalizes the values, and then scales and shifts them using learnable parameters. For more information on normalization click here.

Instance normalization as applied on line 4, 7, 10, 13, 16 and 19 of Listing 6 is used to possibly provide regularization, avoid overfitting, and improve generalization.

Activation functions The ReLU is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. If the input (x) is less than or equal to 0, instead of returning 0 (as in the case of the standard ReLU), the Leaky ReLU returns a small negative value proportional to the input.

A LeakyReLU activation function is used on lines 5, 8, 11, 14, 17 and 20 of Listing 6. A Leaky ReLU allows a small, non-zero gradient (usually a small fraction of the input) for negative inputs. This can help keep those neurons active and learning, mitigating the dead ReLU issue. It is not always the superior choice due to factors as simplicity, empirical performance, and the potential for overfitting.

Dead ReLU A 'dead' ReLU always outputs the same value (zero as it happens, but that is not important) for any input. Probably this is arrived at by learning a large negative bias term for its weights. In turn, that means that it takes no role in discriminating between inputs. Once a ReLU ends up in this state, it is unlikely to recover, because the function gradient at 0 is also 0, so gradient descent learning will not alter the weights. 'Leaky' ReLUs with a small positive gradient for negative inputs are one attempt to address this issue and give a chance to recover.

Convolutional layers The Listing 6 defines a CNN layer not uncommon for character recognition. Its simplicity stems from the relatively low number of output channels. Its simplicity makes training the model less time intensive. In cases where a more complex CNN is needed, it would be possible to extend the simple CNN layer for any model referring to it.

What does it do? The convolution kernel slides over the 2D input array, performs element-wise multiplication and accumulation at each position, and produces a 2D output signal.

64 output channels To actually see what this looks like in practice, follow this link. This webpage shows how a convolutional operation reduces the size of the image. The number of output channels (a parameter on the convolutional layer) defines the number of feature maps a layer will produce. The convolutional layers are defined in Listing 6 on lines 3, 6, 9, 12, 15, 18. Normalization layers are defined on lines 4, 7, 10, 13, 16, 19. Activation functions are defined on lines 5, 8, 11, 14, 17, 20.

Listing 6: CNN in code

```
1  def simple_CNN():
2   simple = AdaptiveCNN(
3   nn.Conv2d(1, 32, kernel_size=(3, 3)),
4   nn.InstanceNorm2d(32),
5   nn.LeakyReLU(),
6   nn.Conv2d(32, 32, kernel_size=(3, 3)),
7   nn.InstanceNorm2d(32),
8   nn.LeakyReLU(),
9   nn.Conv2d(32, 32, kernel_size=(3, 3), stride=2),
10  nn.InstanceNorm2d(32),
11  nn.LeakyReLU(),
12  nn.Conv2d(32, 64, kernel_size=(3, 3)),
13  nn.InstanceNorm2d(64),
14  nn.LeakyReLU(),
15  nn.Conv2d(64, 64, kernel_size=(3, 3)),
16  nn.InstanceNorm2d(64),
17  nn.LeakyReLU(),
18  nn.Conv2d(64, 64, kernel_size=(3, 3), stride=2),
19  nn.InstanceNorm2d(64),
20  nn.LeakyReLU(),
21  )
22  return simple
```

128 output channels In some test conditions RNN-layers will be partly dropped out. When these conditions apply, a CNN with more output channels will be used. The wider CNN looks like in Listing 7, on line 18 the number of output channels are defined (128).

Listing 7: Wider CNN in code

```
1  def advanced_CNN():
2   adv = AdaptiveCNN(
3   nn.Conv2d(1, 64, kernel_size=(3, 3)),
4   nn.InstanceNorm2d(64),
5   nn.LeakyReLU(),
6   nn.Conv2d(64, 64, kernel_size=(3, 3)),
7  nn.InstanceNorm2d(64),
8  nn.LeakyReLU(),
9  nn.Conv2d(64, 64, kernel_size=(3, 3), stride=2),
10 nn.InstanceNorm2d(64),
11 nn.LeakyReLU(),
12 nn.Conv2d(64, 128, kernel_size=(3, 3)),
13 nn.InstanceNorm2d(128),
14 nn.LeakyReLU(),
15 nn.Conv2d(128, 128, kernel_size=(3, 3)),
16 nn.InstanceNorm2d(128),
17 nn.LeakyReLU(),
18 nn.Conv2d(128, 128, kernel_size=(3, 3), stride=2),
19 nn.InstanceNorm2d(128),
20 nn.LeakyReLU(),
21 )
22 return adv
```

Feature maps

For a particular image (Figure 22) feature maps look like Figure 23. In this case 64 feature maps are generated because the last convolutional layer (line 18 in Listing 6) produces 64 output channels.

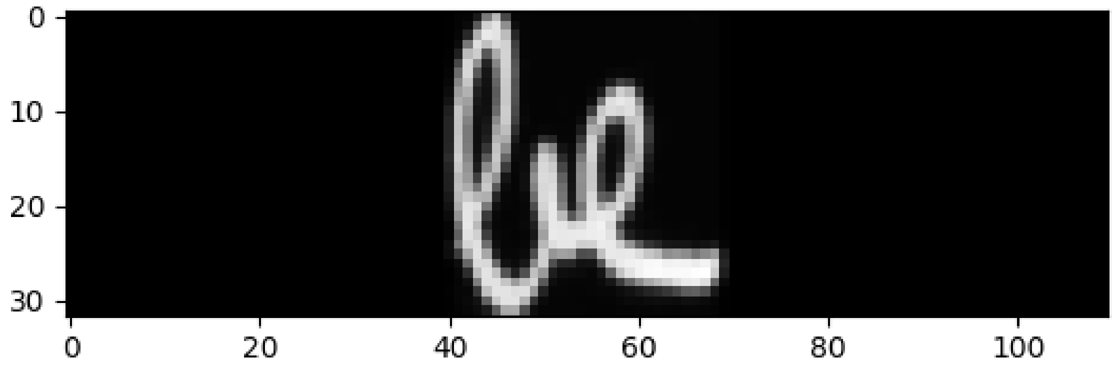


Figure 22: Image representing the word 'be'.

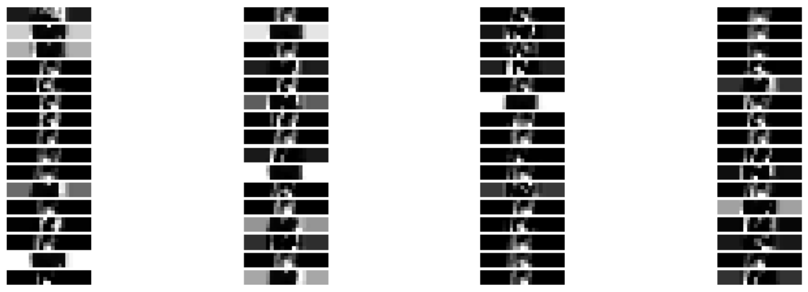
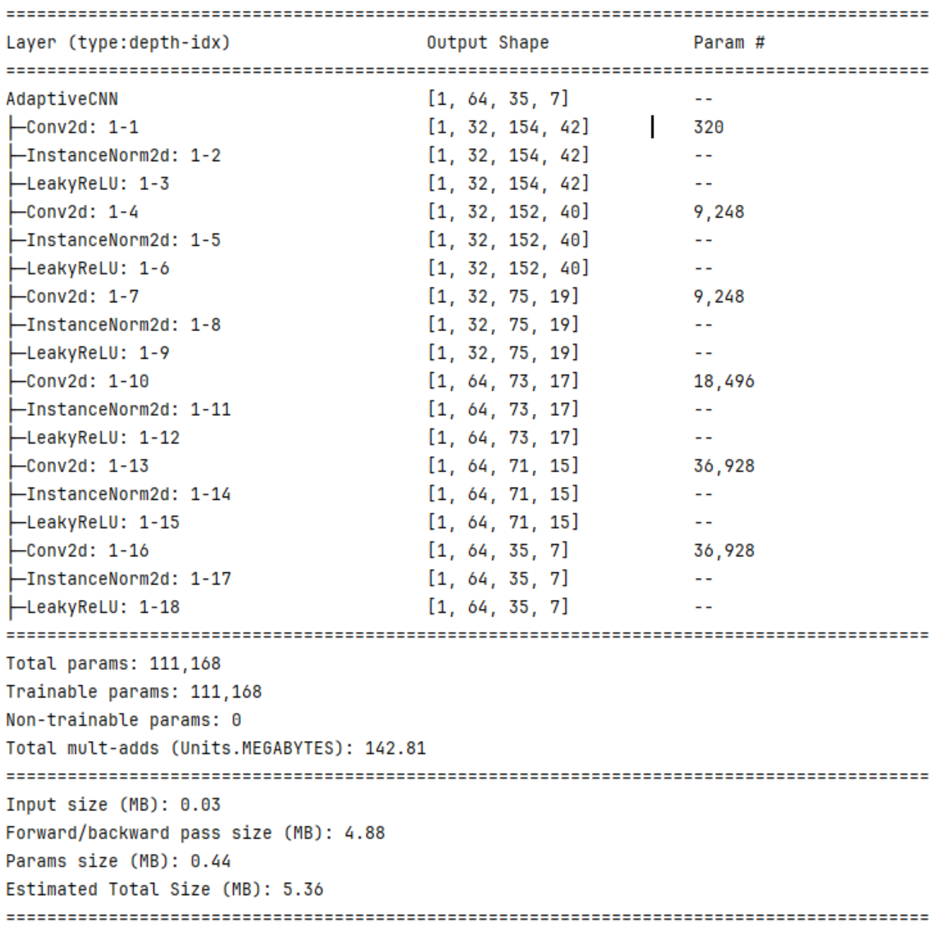


Figure 23: Feature maps for the word 'be' in Figure 22, for every channel an image, totaling 64.

CNN overview

With images having a width of 154 pixels and a height of 44 pixels, the resulting feature map, after applying the complete CNN as defined in Listing 6, contains 64 output channels, each channel having a width of 35 pixels and a height of 7 pixels. The convolutional layer, having a stride of 2, halves size of the image. The layer carrying a depth-idx of 1-7 (in Figure 24) has a stride of 2. After applying this layer the feature map has a size of 75*19 (W X H). The stride halved height of 40 minus padding(2*1) to become 19.



```
=====================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
=====================================================================================
AdaptiveCNN                              [1, 64, 35, 7]            --
├─Conv2d: 1-1                            [1, 32, 154, 42]          320
├─InstanceNorm2d: 1-2                    [1, 32, 154, 42]          --
├─LeakyReLU: 1-3                         [1, 32, 154, 42]          --
├─Conv2d: 1-4                            [1, 32, 152, 40]          9,248
├─InstanceNorm2d: 1-5                    [1, 32, 152, 40]          --
├─LeakyReLU: 1-6                         [1, 32, 152, 40]          --
├─Conv2d: 1-7                            [1, 32, 75, 19]           9,248
├─InstanceNorm2d: 1-8                    [1, 32, 75, 19]           --
├─LeakyReLU: 1-9                         [1, 32, 75, 19]           --
├─Conv2d: 1-10                           [1, 64, 73, 17]           18,496
├─InstanceNorm2d: 1-11                   [1, 64, 73, 17]           --
├─LeakyReLU: 1-12                        [1, 64, 73, 17]           --
├─Conv2d: 1-13                           [1, 64, 71, 15]           36,928
├─InstanceNorm2d: 1-14                   [1, 64, 71, 15]           --
├─LeakyReLU: 1-15                        [1, 64, 71, 15]           --
├─Conv2d: 1-16                           [1, 64, 35, 7]            36,928
├─InstanceNorm2d: 1-17                   [1, 64, 35, 7]            --
├─LeakyReLU: 1-18                        [1, 64, 35, 7]            --
=====================================================================================
Total params: 111,168
Trainable params: 111,168
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 142.81
=====================================================================================
Input size (MB): 0.03
Forward/backward pass size (MB): 4.88
Params size (MB): 0.44
Estimated Total Size (MB): 5.36
=====================================================================================
```

Figure 24: CNN layer overview.

GRU

The model that will be tested contains a GRU-based (Gated Recurrent Unit) RNN. RNNs employing these recurrent units have been shown to perform well in tasks that require capturing long-term dependencies. Visit this site to learn more about RNNs and its occurrences.

To solve the vanishing gradient of a standard RNN, GRU uses an update- and a reset-gate. These two gates decide which information, composed of the previous hidden state and the input for this timestep, should be passed to the output. In this way RNNs can be trained to keep information from earlier timesteps.

Info in this part is all related to Figure 25. It doesn't contain any info on bias variables, even though these parameters exist.

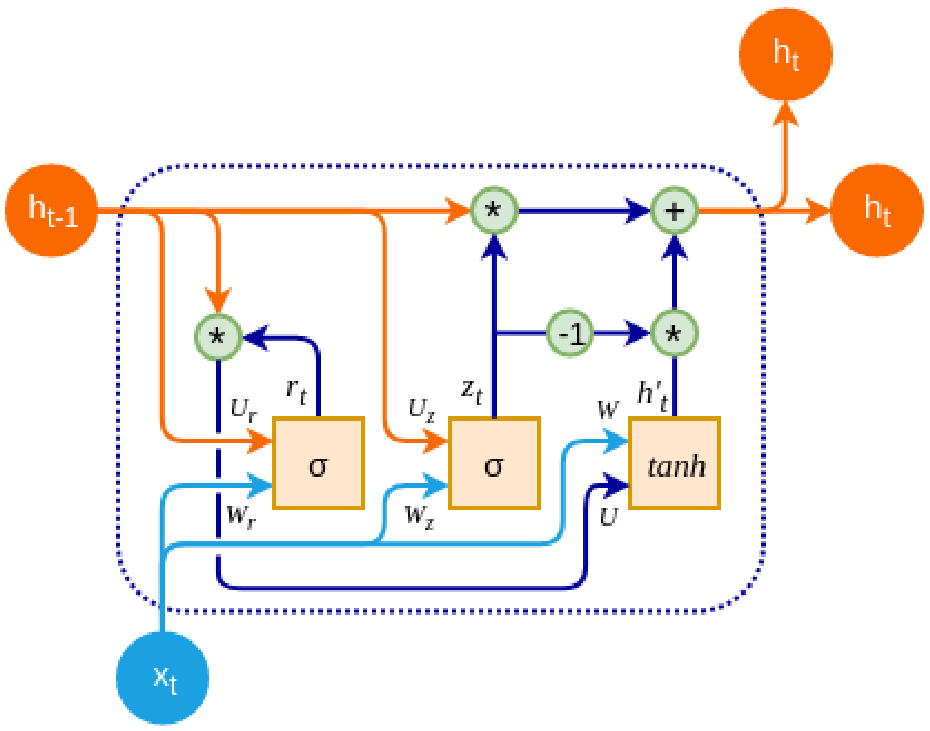


Figure 25: GRU overview.

Some terms used

The outputs of the reset gate r_t and update gate z_t are computed using the current input x_t ,the previous hidden state h_{t-1} and the weight matrices that are learned during training W_r, W_z, U_r, U_z, W, U.

σ -function The σ function is a mathematical function that maps any input value to a value between 0 and 1, making it useful for binary classification and logistic regression problems.

tanh-function The tanh function outputs values in the range of -1 to +1. This means that it can deal with negative values more effectively than the sigmoid function, which has a range of 0 to 1.

Unlike the σ-function, tanh is zero-centered, which means that its output is symmetric around the origin of the coordinate system. This is often considered an advantage because it can help the learning algorithm converge faster.

Hadamard product The element-wise product is usually represented by a $\odot$. Matrices multiplied this way should have the same dimensions

Update gate (z_t)

The update gate determines how much of the new input should be used to update the hidden state. When x_t is plugged into the network unit, it's multiplied by its weight, W_z. The same goes for h_{t-1}, the hidden state from the previous time-step, that gets multiplied by its learned weight (U_z). Both results are added together to and a σ-function is applied to get a result between 0 and 1.

The output of an update gate for a single time-step is computed using the following formula:

$$z_t = \sigma(W_z x_t + U_z h_{t-1})$$

Reset gate (r_t)

The reset gate determines how much of the previous hidden state should be forgotten. The output of the reset gate is computed using the following formula:

$$r_t = \sigma(W_r x_t + U_r h_{t-1})$$

By raising W_r while learning, the current input is given more weight, while lowering U_r will give less weight to the previous hidden state and vice versa.

Current memory content $h_t^{'}$)

The output of the reset-gate influences the product of U and h_{t-1}. So $r_t \odot U$ is responsible for the weight of h_{t-1} in the new hidden state. Furthermore Wx_t, the product of W and x_t is added and a tanh function is applied to get a result between -1 and 1. Using a formula this looks like:

$$h_t^{'} = tanh(Wx_t + r_t \odot Uh_{t-1})$$

New hidden state (h_t)

As a last step, the network needs to calculate the h_t-vector, which holds information for the current unit and passes it down the network. In order to do that the update gate is needed. It determines what to keep from the current memory content $h_t^{'}$ and the hidden state for the previous time-step h_{t-1}. The formula looks as follows:

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h_t^{'}$$

The model can be trained to hold on to information from a previous time-step, and consequently be less interested in newly added information and vice versa. As z_t and $1 - z_t$ complement each other.

Bidirectional RNNs

An architecture of a neural network called a bidirectional recurrent neural network (BRNN) is made to process sequential data. In order for the network to use information from both the past and future context in its predictions, BRNNs process input sequences in both the forward and backward directions.

Dropout

Dropout is a simple yet effective regularization technique. During training, dropout randomly removes a fraction of neurons in the network (in a GRU-based RNN, dropout will be applied to the hidden layer). Typically the dropout rate is between 20% and 50% in each layer. This means that every neuron has a chance of being temporarily 'dropped' (set to zero) during a particular training step. This forces the network to rely on multiple neurons to represent features, rather than depending on a small subset of neurons.