FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Grafeem-naar-foneemconversie door middel van neurale netwerken

Robrecht Meersman
Studentennummer: 01503389

Promotoren: prof. dr. Jan Cnops, mevr. Corinne Bos (Nuance Communications)

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de industriële wetenschappen: informatica

Academiejaar 2018-2019

UNIVERSITEIT
GENT

Woord vooraf

Dit eindwerk was niet mogelijk zonder de steun en de medewerking van een aantal personen.

Tout d'abord, je remercie sincèrement mes superviseurs, Mme Corinne Bos et M. Benjamin Picart, pour leur confiance, leur soutien, leurs conseils et leurs commentaires tout au long de la thèse.

Daarnaast wil ik ook dhr. Bart Baeyens bedanken voor de administratieve ondersteuning en het aanbieden van deze opdracht.
Ik richt eveneens een woord van dank aan Nuance Communications voor het gebruik van de kantoren, de GPU-server, de koffiemachine en vooral alle kennis en hulp van de aanwezige medewerkers.

Ook een woord van dank aan mijn promotor, prof. dr. Jan Cnops, voor de nuttige feedback en extra informatie.
Mevr. Leen Pollefliet verdient ook een woord van dank voor de interactieve lessen en voor haar boek, dat ik meermaals geraadpleegd heb tijdens het schrijven van deze masterproef.

Tot slot wil ik ook mijn vriendin Louise, mijn zus, mijn broer en mijn ouders hartelijk bedanken om mij steeds te steunen in deze periode, alsook voor het nalezen van de scriptie.

Robrecht Meersman

Universiteit Gent - Campus Schoonmeersen,
Valentin Vaerwyckweg 1, 9000 Gent

11/06/2019

Abstract

In deze thesis wordt geëxperimenteerd met verschillende neurale encoder-decoderarchitecturen voor grafeem-naar-foneemconversie. Voorafgaand werk behandelt het converteren van grafeem-naar foneemsequenties als een speciale *neural machine translation*-taak en haalt state-of-the-art resultaten met behulp van convolutionele en *recurrent* attentiongebaseerde modellen. We breiden deze relatie onder meer uit door gebruik te maken van het recent voorgestelde Transformermodel, welke de state-of-the-art resultaten behaalt in machine translation. De modellen worden getraind en geëvalueerd op datasets in drie verschillende talen: de publiek beschikbare CMUDict dataset voor Amerikaans Engels en twee in-house datasets voor Kroatisch en Turks. De voorspellingen worden geanalyseerd en de gemaakte fouten verklaard. Verder exploreren we het grafeem-naar-foneemprobleem door zinnen in hun geheel te converteren, met als doel homografen beter af te handelen.

Trefwoorden — grafeem-naar-foneem, encoder-decoder, attention, RNN, CNN, Transformer

In this paper we experiment with multiple neural encoder-decoder architectures for grapheme-to-phoneme conversion. Previous work has tackled the conversion of grapheme sequences to phoneme sequences as a special case of neural machine translation and achieves state-of-the-art results with convolutional and recurrent attention-based models. We extend this similarity by using the recently purposed transformer model, which achieves results that are closely in line with previous contributions. The models are trained and evaluated on datasets in three different languages: the publicly available CMUDict dataset for American English and two in-house datasets for Croatian and Turkish. Lastly we experiment with a sentence-level approach in order to achieve better homograph conversions.

Index terms — grapheme-to-phoneme, encoder-decoder, attention, RNN, CNN, Transformer

Grapheme-to-phoneme conversion using neural networks

Robrecht Meersman
Supervisors: Jan Cnops (Ghent University), Corinne Bos (Nuance Communications)
robrecht.meersman@ugent.be

Abstract

In this paper we experiment with multiple neural encoder-decoder architectures for grapheme-to-phoneme conversion. Previous work has tackled the conversion of grapheme sequences to phoneme sequences as a special case of neural machine translation and achieves state-of-the-art results with convolutional and recurrent attention-based models. We extend this similarity by using the recently purposed transformer model, which achieves results that are closely in line with previous contributions. Lastly we experiment with a sentence-level approach in order to achieve better homograph conversions.

Index terms — grapheme-to-phoneme, encoder-decoder, attention, Transformer

1 Introduction

Grapheme-to-phoneme conversion (G2P) is the task of converting a written text (graphemes) into its pronunciations (phonemes). For example, when given the word *able*, the task is to find the correct transcription, which in this case is `EY B AH L`. Note that some cases are ambiguous, e.g. *lead* can be `L IY D` or `L EH D` depending on the meaning of the word in the phrase. G2P is the most important component of concatenative text-to-speech systems and still has room for improvement.

Earlier G2P systems use a lexicon of orthography-transcription pairs and a large chain of grapheme-to-phoneme rules for handling out-of-vocabulary words. Developing such rules is language-dependent and requires linguistic expertise. Due to propagation in that chain of modules, context information is sometimes lost and an incorrect phoneme sequence is assigned.

Other approaches use *joint-sequence models* [2, 3, 4], which align graphemes to phonemes (creating *graphones*) in order to construct an n-gram model. This models the probability of a graphone given the n previous graphones. However, joint-sequence models require a dataset in which the alignment of graphemes and phonemes is annotated, which again requires additional linguistic knowledge.

With the increasing speed of GPUs and the growing accessibility of data, it is possible to build an artificial neural network which learns to convert graphemes to phonemes when given enough examples. Previous research indicates that the use of neural networks has great potential on the G2P-task [2, 13, 15, 18, 19]. Because the neural networks learn the linguistic rules themselves, they do not need to be manually inserted by experts, thus a prototype can be used in production more quickly.

In essence the G2P-task is similar to Machine Translation. A word from a source language can correspond to one or more words from the target language, which do not have to be in the same order. Instead of converting a sequence of words out of a (finite) vocabulary, we convert a sequence of characters out of a finite alphabet.

In this paper we experiment with different attention-based neural architectures on the problem of grapheme-to-phoneme conversion. We experiment with the best performing topologies of previous work, as well as introduce the transformer architecture. We will also experiment with sentence-level G2P and train neural networks on complete sentences in order to improve homograph accuracy.

2 Models

2.1 RNN-based Models

In its most basic form, an RNN (Recurrent Neural Network) is a multilayer perceptron with 1 hidden layer where, together with the input vector $\mathbf{x}_t$, the internal state of the previous iteration $\mathbf{h}_{t-1}$ is used as input. The hidden state $\mathbf{h}$ at step t is calculated in the following way:

$$\mathbf{h}_t = g(W\mathbf{x}_t + U\mathbf{h}_{t-1} + \mathbf{b}_h),$$

where W, U and $\mathbf{b}_h$ are trainable variables and g is an activation function. RNNs are very useful for time-based tasks or sequence-to-sequence problems. The Long Short-Term Memory (LSTM) cell, a variant on an RNN cell, is particularly useful for modelling long-term dependencies.

Encoder-Decoder LSTM

Encoder-decoder networks have been widely used for sequence-to-sequence type problems. Our architecture follows [14] and [18] and is depicted in Figure 1. The encoder-decoder model uses two (deep) LSTMs: (1) The encoder bidirectional LSTM stack transforms the embedded orthography

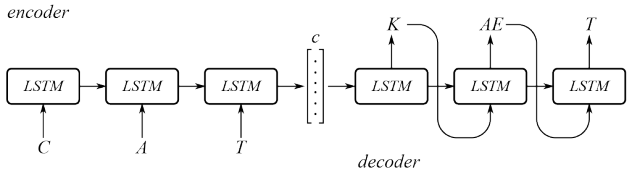


Figure 1: An encoder-decoder LSTM transcribes the word K AE T given the context vector $\mathbf{c}$.

sequence into a fixed size context vector $\mathbf{c}$. (2) The decoder LSTM stack produces a phoneme sequence conditioned by that context vector.

In a bidirectional LSTM (BLSTM), one cell processes the input from left to right, while the other processes input from right to left. The two outputs are then combined in a trainable layer [6]:

$$\overrightarrow{\mathbf{h}_t} = \mathrm{LSTM}(\mathbf{x}_t, \overrightarrow{\mathbf{h}_{t-1}})$$
$$\overleftarrow{\mathbf{h}_t} = \mathrm{LSTM}(\mathbf{x}_t, \overleftarrow{\mathbf{h}_{t+1}})$$
$$\mathbf{h}_t^{[l]} = W_1\overrightarrow{\mathbf{h}_t} + W_2\overleftarrow{\mathbf{h}_t} + \mathbf{b},$$

with $\mathbf{h}_t^{[l]}$ being the hidden state of the l-th layer, on step t. $\mathrm{LSTM}(\cdot)$ refers to the computations of a single LSTM cell. The input sequence $\mathbf{x}_t$ is represented by a character embedding: a one-hot encoded vector of the input tokens, multiplied by a jointly trained weight matrix.

In a stacked LSTM, the output of an LSTM layer is fed into another LSTM, using:

$$\mathbf{h}_t^{[1]} = \mathrm{LSTM}_1(\mathbf{x}_t, \mathbf{h}_{t-1}^{[1]})$$
$$\mathbf{h}_t^{[l]} = \mathrm{LSTM}_l(\mathbf{h}_t^{[l-1]}, \mathbf{h}_{t-1}^{[l]}),$$

with $l > 1$. Stacking multiple LSTMs makes the network better at generalizing higher-level dependencies.

Using the previously predicted phonemes $\hat{y}_{0:t-1}$ (with a *start-of-sequence* token at $t = 0$) and the context vector $\mathbf{c}$, the decoder outputs a probability distribution over all phonemes:

$$\mathbf{p}(y_t|\hat{y}_{0:t-1}, \mathbf{c}) = \mathrm{softmax}(V \cdot \mathbf{h}_t + \mathbf{b}),$$

with $\mathbf{h}_t$ the output of the last decoder LSTM layer and V, $\mathbf{b}$ trainable parameters. Using beam search or argmax (greedy search), a phoneme sequence can be inferred. The decoding process stops when a special *end-of-sequence* token is predicted. For faster and more stable convergence, we use the ground truth $y_{0:t-1}$ during the training phase instead of the previously predicted tokens $\hat{y}_{0:t-1}$ in a mechanism called *teacher forcing*.

Attention-based Encoder-Decoder LSTM

The encoder-decoder LSTM is very powerful when alignment of grapheme and phoneme pairs is not available [18]. However, the network performs badly on long words, because more information must be stored in the fixed-size context vector.

An extension to the LSTM encoder-decoder initially proposed by [1] and later improved by [11], introduces an attention mechanism. The goal is to modify the context vector $\mathbf{c}$

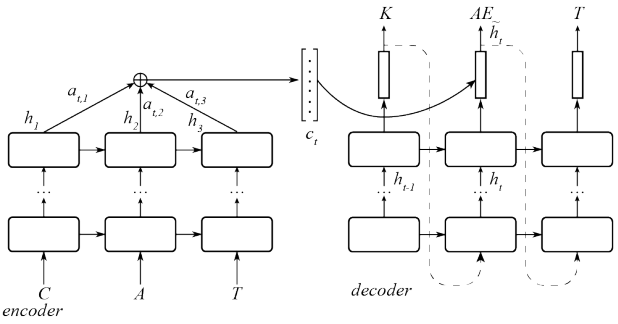


Figure 2: Encoder-decoder LSTM with global Luong attention.

for every output label prediction. Instead of representing the entire sequence with the last hidden state, all hidden states of the encoder are combined to generate a context vector $\mathbf{c}_t$ for each prediction. There are two types of attention: global attention and local attention. The mechanism is roughly the same, with the difference that global attention has to attend all hidden states, where local attention only has access to a limited window in order to reduce the overhead for long input sequences. Since orthographies have a relatively short length, we will only discuss global attention. Figure 2 depicts a global attention mechanism as proposed by [11].

The encoder is not much different from a model without attention, except that all hidden states $\mathbf{h}_s$ are now kept. The decoder works slightly different. At each decoding step t, a context vector $\mathbf{c}_t$ is computed based on a weighted sum of all the hidden states $\mathbf{h}_s$ of the encoder:

$$\mathbf{c}_t = \sum_s a_{t,s}\mathbf{h}_s,$$

with $a_{t,s}$ being the attention weights. The context vector $\mathbf{c}_t$ is then used to get the final output scores $\tilde{\mathbf{h}}_t$. There are two ways in which the context vector can be combined with the output of the decoder:

$$\tilde{\mathbf{h}}_t = \begin{cases} \mathrm{RNN}([\mathbf{c}_t; \mathbf{h}_{t-1}]) & \textit{Bahdanau} \\ \tanh(W_c[\mathbf{c}_t; \mathbf{h}_t]) & \textit{Luong}. \end{cases}$$

The attention weights $a_{t,s}$ are computed via a softmax of attention *scores* between the encoder outputs $\mathbf{h}_s$ and the decoder state $\mathbf{h}_t$. This results in an attention vector $\mathbf{a}_t$, containing an attention score $a_{t,s}$ for each $\mathbf{h}_s$:

$$a_{t,s} = \frac{\exp\left(\mathrm{Score}(\mathbf{h}_t, \mathbf{h}_s)\right)}{\sum_S \exp\left(\mathrm{Score}(\mathbf{h}_t, \mathbf{h}_s)\right)}.$$

The scores also have different options:

$$\mathrm{Score}(\mathbf{h}_t, \mathbf{h}_s) = \begin{cases} \mathbf{h}_t^T\mathbf{h}_s & \textit{dot} \\ \mathbf{h}_t^T W_a \mathbf{h}_s & \textit{general} \\ \mathbf{v}_a^T \tanh(W_a[\mathbf{h}_t; \mathbf{h}_s]) & \textit{concat.} \end{cases}$$

The weights W_c, W_a en $\mathbf{v}_a$ are trained jointly with the encoder-decoder, resulting in a soft-alignment between graphemes and phonemes.

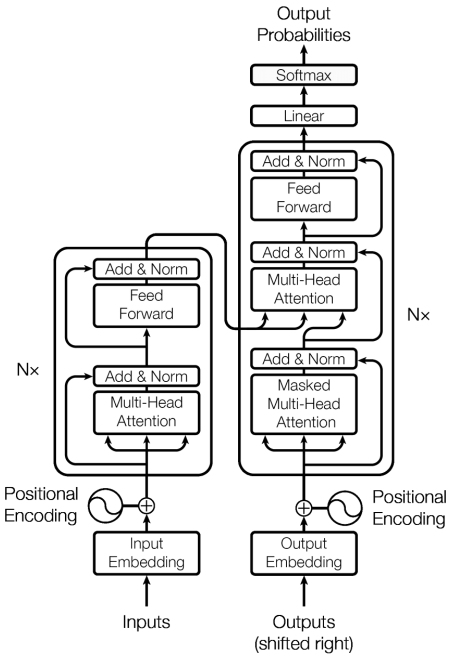


Figure 3: An illustration of the Transformer [16]. The encoder cells (left) and decoder cells (right) are both repeated N times.

2.2 CNN-based Attention Model

RNN-based models with attention have proven to be very powerful for G2P and other NLP problems. However, they do not lend themselves easily to parallelization, the strong point of moderns GPUs. LSTMs are also very computationally intensive. The convolutional sequence-to-sequence model of [5] attempts to parallelize encoding as much as possible by replacing RNN cells with a convolutional neural network (CNN). By stacking the convolutional blocks, both local and long-range relationships can be modeled. The encoder performs multiple 1D convolutions over the entire input sequence at once to create encoder states $\mathbf{h}_s$. The decoder CNN predicts output probabilities by using its hidden state $\mathbf{h}_t$ and the encoder states $\mathbf{h}_s$ to create an attention vector $\mathbf{c}_t$ for each step t.

2.3 Transformer Model

The recently purposed *Transformer* architecture [16] tries to tackle both parallelization and long-range dependencies by completely omitting the recurring aspect and merely modeling the dependencies with a concept of self-attention. It achieves state-of-the-art results in Machine Translation, so it can be worthy to experiment with in the G2P domain. Figure 3 shows a simplified representation of the Transformer, as presented in [16]. Like previously discussed models, the Transformer also makes use of an encoder-decoder structure.

Encoder

The encoder is a stack of N encoder blocks. The first block receives the character embedded vector of the input sequence, the others an increasingly more abstract version thereof. The idea here is that closely spaced elements interact with each other in the lower blocks to find the local dependencies, while long-range dependencies are captured in the higher blocks.

The encoder blocks themselves consist a multi-headed self-attention layer, which calculates a self-attention vector $\mathbf{z}_i$ for each token i in the sequence, and a feedforward layer with residual connections. The calculation of the self-attention vector $\mathbf{z}_i$ is done in two steps.

In a first step, a query vector $\mathbf{q}_i$, key vector $\mathbf{k}_i$ and value vector $\mathbf{v}_i$ are calculated for each input token i (simultaneously by using matrices) by multiplying the embedded input sequence X with three different weight matrices:

$$Q = XW_Q$$
$$K = XW_K$$
$$V = XW_V$$

The second step calculates a self-attention vector Z for each pair of input tokens, based on an attention score for the query and key vectors of each pair. The attention vector is calculated on all queries $\mathbf{q}_i$ simultaneously:

$$Z = \text{Attention}(Q, K, V)$$
$$= \text{softmax}(\frac{Q \cdot K^T}{\sqrt{d_k}})V,$$

with d_k the encoding dimension of K. This score determines how related a symbol is to another symbol in the sequence.

To make the self-attention layer even more powerful, an extra multi-headed mechanism is added. Instead of calculating one attention matrix, h different versions of Z are calculated, each with their own weight matrices. These attention heads are concatenated to each other to obtain one wide multi-headed attention vector M, which is compacted and sent to the higher encoder blocks or to the decoder.

Decoder

The structure of the decoder is very similar to that of the encoder, but it has an extra layer between the self-attention and the feedforward layer. First it attends the previously generated phoneme output in a self-attention layer. Next, the self-attention is combined with the output of the encoder in a second attention layer.

3 Experiments

3.1 Data

For the experiments we make use of four datasets. The first is the CMUDict[1] dataset, a public dataset which contains 126,191 American English words with their transcription. This dataset is chosen in order to directly compare with earlier results. As in [4, 13, 15, 18, 19], a training set of 113,438 and a test set of 12,000 words are used. 5,660 samples are held back from the training set as validation. The transcriptions use the ARPABET, a minimal ASCII phoneme alphabet for American English. Primary and secondary stress suffixes are omitted. This results in a grapheme alphabet of 27 characters (all alphabet tokens and an apostrophe) and a phoneme alphabet of 39 tokens.

[1] We are grateful to Stan Chen for providing the data.

The other two word-level datasets are in-house datasets of Nuance® Communications[2]. They make use of a more complex, IPA-like alphabet, containing many suprasegmentals such as stress, syllable boundaries and tone contours. We use a dataset of 106,457 Turkish words and a set of 109,006 Croatian words. The data is split into train-test-val splits using ratios of 80% - 10% - 10% and bucketing is used to create minibatches of similar sequence length.

The last dataset contains transcriptions at sentence-level. From a series of American English corpuses 50,000 sentences are randomly selected and supplemented by specifically chosen sentences that each contain a homograph (i.e. an orthography with two pronunciations). This results in a corpus of 67,998 sentences (1,524,430 words, of which 65,528 different orthographies).

3.2 Models

The RNN architecture uses the "Tensorflow NMT toolkit" [10]. This toolkit is intended for experimentation with various attention mechanisms in Machine Translation tasks. The toolkit has multiple attention mechanisms available and hyperparameters can be adjusted easily. We can consider G2P as a special MT problem and add custom metrics for word and phoneme error rates.

For the attention-based CNN architecture, the NMT toolkits functionality is extended to serve this purpose. The implementation follows the architecture described in [5].

The Transformer model uses "g2p-seq2seq"[3] developed at CMU. Results on CMUDict are mentioned in the repository, but – to our knowledge – no publications nor thorough experiments on this newly proposed architecture have been made.

3.3 Training

All models are trained on an NVIDIA® Tesla® K80 GPU for a maximum of 200K steps until there is no improvement measured on the validation set. Training takes approximately 4 hours per model.

A dropout layer is provided between each layer. We use a dropout rate of 0.2 (keep probability 0.8) for all models. We use Adam as an optimizer, with epsilon values of $\epsilon = 10^{-4}$ for the RNN model, $\epsilon = 10^{-3}$ for the CNN model and $\epsilon = 10^{-6}$ for the Transformer.

3.4 Inference

A beam search decoder is used for evaluation, with beam width $k = 5$ in the RNN model and greedy decoding ($k = 1$) for the CNN model. In the Transformer architecture we experiment with the impact of the beam width on the error rate.

For evaluation metrics we calculate the Word Error Rate (WER) and Phoneme Error Rate (PER). The WER is the ratio of wrongly predicted words over the total amount of words. The PER measures the amount of phoneme-level mistakes divided by the total amount of phonemes in the ground truth. The mistakes are computed as the edit distance (Levenshtein distance): the amount of operations (insertion, deletion, replacement of a phoneme) needed to transform the predicted sequence into the expected sequence. When converting graphemes-to-phonemes on a word-level basis, the model has no sentence context for transcribing homographs correctly. Therefore, we tolerate all pronunciations of an orthography. The PER will use the pronunciation with the smallest edit distance.

[2]We are grateful to Nuance® Communications for providing the data, intelligence and infrastructure.

[3]https://github.com/cmusphinx/g2p-seq2seq

Table 1: Comparison of the best results for CMUDict between different topologies.

Model	**PER** (%)	**WER** (%)
RNN-based	6.00	24.50
CNN-based	7.22	28.76
Transformer	**4.73**	**20.24**

Table 2: The impact of the number of layers l, the number of hidden units d and the attention mechanism on the PER and WER for an RNN-based network on CMUDict 0.7b.

l	d	**Attention**	**PER** (%)	**WER** (%)
3	256	No attn.	9.9	39.4
	512		10.8	41.0
		Bahdanau	10.1	40.7
		Luong	9.4	37.2
2	256		12.7	46.1
	512		9.2	37.3
	1,024		**9.0**	**36.7**

4 Results

4.1 Experiments on Word-Level

First, we experiment on a word-level basis, training each model with a vocabulary of words and their transcription. The best PER and WER for the three mentioned topologies are depicted in Table 1. The Transformer model achieves the best results, followed by the RNN. When training on the in-house datasets, we see similar figures. Depending on the amount of samples, the used phoneme alphabet and the phonemic characteristics of the writing system, the error rates can be significantly different, but the relative performance differences between the models remain the same.

For the RNN-based model we experiment with the amount of layers l, the amount of hidden units d and the attention mechanism used. The results shown in Table 2 are obtained using version 0.7b of CMUDict, in which stress suffixes were not removed. We see that the layer amount has less effect on the accuracy when compared to the amount of units and the attention mechanism used. Accuracy increases with the number of units, but so do inference time and model size (not shown in the table).

For the CNN-based model, the impact of the number of layers on the PER and WER is examined and presented in Table 3. The training phase of the CNN-based model is to our experience very unstable when hyperparameters are used

Table 3: The impact of the amount of layers l on the PER and WER of a CNN-based model on CMUDict.

l	**PER** (%)	**WER** (%)
3	24.89	59.98
4	**7.22**	**28.76**
5	26.18	67.86

Table 4: The impact of the number of layers l, the number of hidden units d and the number of attention heads h on the PER and WER for a Transformer network on in-house Turkish data.

l	d	h	**PER** (%)	**WER** (%)	**Size** (MB)	**Speed** (ms)
2	128	1	4.79	23.29	**3.6**	**41.2**
		2	4.29	20.73	**3.6**	44.1
3	256	4	3.30	15.50	15.1	80.5
	512		16.54	56.69	48.3	160.8
4	256		3.07	14.53	20.2	81.7
5	256		**3.04**	**14.50**	25.2	104.0
	512		43.49	93.25	80.4	-
6	512	8	53.99	96.76	96.4	-

that are different from $l = 4$, $d = 256$ and filter width $k = 3$. Models with $d > 256$ fail to converge.

For the Transformer we experiment with some parameters as well: the number of layers l, the number of hidden units d and the amount of attention heads h. The experiments are performed on the in-house dataset of Turkish words. In addition to the PER and WER, the model size and the speed are also presented in Table 4. The speed is measured for inference of one word on CPU and is averaged over 200 words. The accuracy of the model increases with the number of trainable parameters, but a maximum is achieved with 4 layers. We see that a hidden dimension of $d > 256$ consistently results in a poor accuracy.

Finally, our best model is compared with previous results on the CMUDict dataset. Table 5 shows that our Transformer model is competitive to the most recent work delivered in the G2P domain.

Beam Width on Error Rate

The influence of the beam width k during inference is also analyzed and presented in Table 6. There is a small advantage when compared to greedy search ($k = 1$) and there is no improvement for $k > 2$.

Phoneme-Level Error Comparison by Category

It may be interesting to know exactly where the predictions of a neural network go wrong. To find out, we look at the type of phoneme-level mistakes made by the Transformer model. The type of mistake (replacement, deletion or insertion of a phoneme) can be obtained with a backtrace matrix in the edit distance algorithm. The mistakes made are divided into categories and their share is computed. Table 7 shows the most common mistakes for the Turkish language, which are mostly suprasegmentals. Over- and underprediction are a common problem in neural G2P and MT, caused by the model failing

Table 5: Comparison of previous results on CMUDict to our best model (Transformer, 3 layers 256 dimensions 4 attn.-heads, beam-width 2).

Model	**PER** (%)	**WER** (%)
LSTM with full delay [13]	9.11	30.1
Joint sequence model [2]	5.88	24.53
Encoder-decoder LSTM [18]	7.63	28.61
Enc.-dec. LSTM w. global attention [15]	5.04	21.69
Ensemble of 5 models [15]	4.69	20.24
Encoder CNN, decoder BLSTM [19]	4.81	25.13
Transformer,($l = 4$, $d = 256$, $h = 8$)	**4.73**	**20.24**

Table 6: The effect of beam width k on PER and WER. Performed on the best Transformer model for CMUDict.

k	**PER** (%)	**WER** (%)
1	4.91	20.48
2	4.86	20.42
3	4.87	20.42
4	4.87	20.42

to attend a character or attending it twice. Note that these figures differ a lot when compared across languages, since they are essentially defined by the phonemic characteristics of a writing system.

Error Comparison by Word Length

As in [15, 19], we split the words of the CMUDict dataset into four categories based on the length of the orthography: *short* (length ≤ 6), *medium* (length $\in \{7, 8\}$), *long* (length $\in \{9, 10\}$) and *very long* (length ≥ 11). In Figure 4 we see a small global rise in WER as the words get longer. The RNN-based model has the largest increase of WER and the Transformer model the smallest. This can be explained by the fact that RNNs are biased towards neighboring cells and therefore fail to model long-range dependencies, despite the addition of memory cells in LSTMs. These deficiencies have been pointed out by [7], [8] and [9].

Table 7: Most common mistakes by the Transformer model on in-house Turkish data.

Error	**Share in PER** (%)
Stress	59.39
Vowel duration	10.30
Incorrect vowel	5.76
Underprediction	5.15
Incorrect consonant	4.55
Overprediction	3.64
Syllable boundaries	2.42
Other	8.79

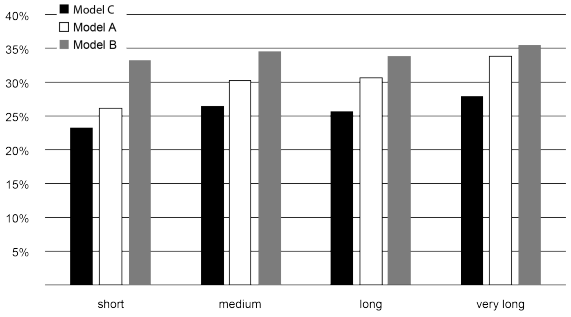


Figure 4: Comparison of WER in function of the word length for the RNN-based model (Model A), the CNN-based model (Model B) and the Transformer (Model C).

Table 8: Results of a sentence-level model on homographs.

Orth.	**Transcriptions**	**Accuracy** (%)
bow	[boʊ], [baʊ]	68.1
content	[ˈkɑn.tɛnt], [kən.ˈtɛnt]	85.1
lead	[ˈlɛd], [ˈlid]	50.1
live	[ˈlaɪv], [ˈlɪv]	90.4
present	[ˈprɛ.zənt], [prɪ.ˈzɛnt]	87.4
read	[ˈrɛd], [ˈrid]	62.3
record	[ˈrɛ.kərd], [rɪ.ˈkɔrd]	83.7
subject	[ˈsʌb.dʒekt], [səb.ˈdʒekt]	90.9
tear	[ˈter],[ˈtir]	90.3
tears	[ˈterz],[ˈtirz]	69.5
used	[ˈjust], [ˈjuzd]	94.1
wind	[ˈwaɪnd], [ˈwɪnd]	91.2
winds	[ˈwaɪndz], [ˈwɪndz]	87.3
Average accuracy		80.8 ± 13.7

4.2 Experiments on Sentence-Level

Next a Transformer model is trained at sentence-level. Because we want the neural network to learn relationships across different words, we train with a larger Transformer model of 6 layers, 512 hidden units and 8 attention heads. After training, the model is evaluated on a series of sentences that each contain a known homograph. A beam width $k = 2$ is used for inference. The accuracy on the homographs is shown in Table 8.

Another Transformer network was trained on a word-level lexicon and managed to convert all homographs present in the table to one of the two transcriptions. Thus, an ideal word-level model will have an average homograph accuracy of 50%.

With an average homograph accuracy of 80.8%, we can conclude that a model trained on sentence-level performs better in homograph transcription when compared to a model that is trained on word-level.

5 Conclusion

After thorough experimentation with different neural architectures, the Transformer architecture delivers the best Word and Phoneme Error Rates on different languages. The results are closely in line with previous works in the G2P domain.

All models succeed well in learning the many-to-many alignments and choose the correct vowels and consonants with good reliability. The greatest difficulties encountered by the models are suprasegmentals, such as stress and vowel length. Whereas previously proposed architectures have more difficulty with longer words, the Transformer architecture appears to be more robust against this problem.

When trained on sentence-level, the Transformer model is also more capable of predicting the correct homograph transcription.

Although in practice rule-based models still give a lower error rate, they require linguistic knowledge and large start-up costs before a first prototype can be used. Compared to these rule-based models, neural models achieve good accuracy in a very short time without the intervention of an expert. They can easily be put into production as a prototype for a new language, until a rule-based system exceeds its performance. Both neural and rule-based systems can also be combined for better handling of out-of-vocabulary words.

5.1 Future Work

To date, models that perform well in machine translation are also ideal for grapheme-to-phoneme conversion when alignment of grapheme and phoneme pairs is not present in the data. However, if we look at the G2P task within the context of TTS, the conversion of graphemes to phonemes is only an intermediate step towards the final goal: to generate a sound wave given an orthography input. Various proposals have been made to treat a TTS system as an end-to-end network [12, 17]. They do not make use of the fairly limited phoneme representation, but learn an implicit G2P model, in which prosody and emotion can be modeled better. These systems come with a drawback that their internal representation is not legible, making it more difficult to analyze. A second limitation is the lack of good data. An end-to-end system requires a large amount of recorded sound clips from a professional speaker. Although these systems still have many drawbacks, end-to-end approaches are promising and already rising in domains such as speech recognition and autonomous driving.

References

[1] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[2] Bisani, M. and Ney, H. (2008). Joint-sequence models for grapheme-to-phoneme conversion. *Speech communication*, 50(5):434–451.

[3] Chen, S. F. (2003). Conditional and joint models for grapheme-to-phoneme conversion. In *INTERSPEECH*.

[4] Galescu, L. and Allen, J. F. (2002). Pronunciation of proper names with a joint n-gram model for bi-directional grapheme-to-phoneme conversion. In *Seventh International Conference on Spoken Language Processing*.

[5] Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. (2017). Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1243–1252. JMLR. org.

[6] Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE.

[7] Lai, S., Xu, L., Liu, K., and Zhao, J. (2015). Recurrent convolutional neural networks for text classification. In *Twenty-ninth AAAI conference on artificial intelligence*.

[8] Linzen, T., Dupoux, E., and Goldberg, Y. (2016). Assessing the ability of lstms to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4:521–535.

[9] Liu, F., Baldwin, T., and Cohn, T. (2017). Capturing long-range contextual dependencies with memory-enhanced conditional random fields. *arXiv preprint arXiv:1709.03637*.

[10] Luong, M., Brevdo, E., and Zhao, R. (2017). Neural machine translation (seq2seq) tutorial. *https://github.com/tensorflow/nmt*.

[11] Luong, M.-T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

[12] Ping, W., Peng, K., Gibiansky, A., Arik, S. Ö., Kannan, A., Narang, S., Raiman, J., and Miller, J. (2017). Deep voice 3: 2000-speaker neural text-to-speech. *CoRR*, abs/1710.07654.

[13] Rao, K., Peng, F., Sak, H., and Beaufays, F. (2015). Grapheme-to-phoneme conversion using long short-term memory recurrent neural networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4225–4229. IEEE.

[14] Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

[15] Toshniwal, S. and Livescu, K. (2016). Jointly learning to align and convert graphemes to phonemes with neural attention models. In *2016 IEEE Spoken Language Technology Workshop (SLT)*, pages 76–82. IEEE.

[16] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

[17] Wang, Y., Skerry-Ryan, R., Stanton, D., Wu, Y., Weiss, R. J., Jaitly, N., Yang, Z., Xiao, Y., Chen, Z., Bengio, S., et al. (2017). Tacotron: A fully end-to-end text-to-speech synthesis model. *arXiv preprint arXiv:1703.10135*.

[18] Yao, K. and Zweig, G. (2015). Sequence-to-sequence neural net models for grapheme-to-phoneme conversion. *arXiv preprint arXiv:1506.00196*.

[19] Yolchuyeva, S., Németh, G., and Gyires-Tóth, B. (2019). Grapheme-to-phoneme conversion with convolutional neural networks. *Applied Sciences*, 9(6):1143.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Lijst van afkortingen

BLEU	BiLingual Evaluation Understudy
BLSTM	Bidirectional Long Short-Term Memory network
BPTT	Back-Propagation Through Time
CNN	Convolutional Neural Network
G2P	Grapheme-to-Phoneme
GLU	Gated Linear Unit
IPA	International Phonetic Alphabet
LCS	Longest Common Subsequence
LSTM	Long Short-Term Memory network
MT	Machine Translation
NLP	Natural Language Processing
NP	Nondeterministic Polynomial
PCA	Principale Componenten-Analyse
PE	Positional Encoding
PER	Phoneme Error Rate
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
WER	Word Error Rate

1 Inleiding

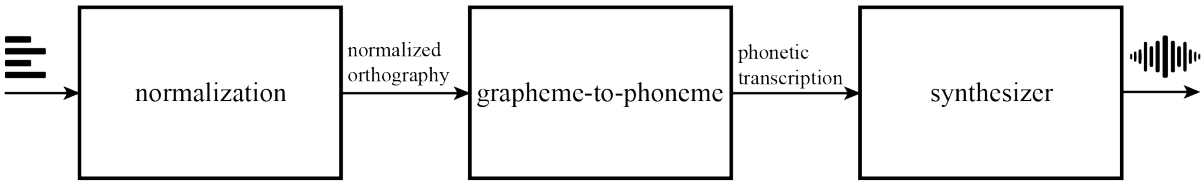
Grafeem-naar-foneemconversie (G2P, *grapheme-to-phoneme*) zoekt naar de uitspraak van een woord gegeven zijn geschreven vorm. Bijvoorbeeld wanneer het Engelse woord "able" gegeven wordt, moet de correcte uitspraak `EY B AH L` gevonden worden. De uitspraak van letters (vooral van klinkers) is bij veel talen niet eenduidig en is afhankelijk van de omliggende letters. Merk op dat zelfs woorden niet eenduidig omgezet kunnen worden, bijvoorbeeld het woord "lead" kan zowel als `L IY D` of als `L EH D` uitgesproken worden, afhankelijk van de betekenis van het woord in de zin. Deze woorden worden *homografen* genoemd.

G2P is de hoofdmodule van text-to-speech (TTS), de technologie die digitale tekst omzet naar een spraaksignaal. TTS en andere technologieën uit het *Natural Language Processing*-domein (NLP) blijven de laatste jaren steeds verbeteren, maar blijven een onafgewerkt hoofdstuk. Je vindt TTS-systemen terug in GPS-software, in de medische sector als hulp bij mensen met een visuele of spraakbeperking, in treinstations als automatische aankondigingen en in de steeds populairder wordende digitale assistenten zoals de Google Assistent, Apples Siri en de Amazon Alexa. Het einddoel voor TTS is een model creëren dat eender welke zin correct en verstaanbaar kan uitpreken, en idealiter niet meer te onderscheiden is van een opname van een menselijke stem.

Deze thesis gebeurt in samenwerking met Nuance Communications. Nuance is een Amerikaans bedrijf dat zich voornamelijk bezighoudt met de ontwikkeling van spraakherkenningssoftware en scantoepassingen. Daarnaast heeft het ook een afdeling voor TTS, met R&D-teams voor onderzoek op zowel korte als lange termijn.

1.1 Opbouw van een TTS-systeem

Een TTS-systeem kan grotendeels opgedeeld worden in drie modules die sequentieel worden uitgevoerd, zoals weergegeven in Figuur 1.
De eerste module voert de **text normalization** uit. Dit komt neer op het voluit schrijven van alle afkortingen en getallen, op de manier waarop ze uitgesproken worden. Zo wordt "Dr." omgezet naar "doctor". Text normalization kan geïmplementeerd worden via reguliere expressies, maar die houden geen rekening met context (zo kan "St." bijvoorbeeld "Saint" of "Street" betekenen). Statistische benaderingen zoals Hidden Markov Models berekenen dan weer de kans op elke normalisatie.
De tweede module voert de **grafeem-naar-foneem** (G2P, *letter-to-sound*) conversie uit. Hierbij wordt de genormaliseerde tekst (*orthografie*) omgezet naar een fonetische transcriptie. Een fonetisch schrift gebruikt een alfabet van atomaire klanken die kunnen gebruikt worden om de uitspraak van een woord te beschrijven.
De derde module, de **synthesizer** of back-end, zet de foneemsequentie om naar geluidsgolven. Hiervoor bestaan er verschillende technieken, met elk hun voor- en nadelen. De populairste techniek werkt met een database bestaande uit opnames van een professionele spreker. Vervolgens worden de geluidsfragmenten gesplitst in koppels van twee fonemen. Om een woord te construeren worden de verschillende korte geluidsfragmenten aan elkaar geconcateneerd met een overlap van één foneem. Andere technieken genereren de golfvormen van nul af aan met behulp van een akoestisch model en manueel ingebrachte regels per foneem. Deze benadering resulteert vaak in een *robotische* stem, maar ze heeft geen nood aan een grote databank van opnames. Deze geheugenoptimalisatie maakt ze vooral interessant voor mobiele apparaten zoals een e-reader of GPS. De nieuwste technieken maken gebruik van neurale netwerken en benaderen een klank die



Figuur 1: Blokschema van een TTS-systeem.

evenwaardig is aan een menselijke stem (Van Den Oord et al., 2016).

1.2 Fonemen en foneemalfabetten

Fonemen zijn per definitie "een verzameling klanken die een betekenisonderscheidende functie hebben". Ze worden vaak gecategoriseerd op basis van de manier waarop ze geproduceerd worden, zoals de positie van de tong en de manier van luchtstroom. Het IPA (International Phonetic Alphabet) is hierbij de algemene standaard sinds de 19e eeuw. Naast de 107 letters voor klinkers en medeklinkers, kunnen ook 31 diakritische tekens gebruikt worden om hun uitspraak licht te wijzigen. IPA biedt ook nog 19 extra suprasegmentale symbolen aan om lengte, toonhoogte, klemtoon en intonatie aan te duiden. Het woord "excusing" wordt bijvoorbeeld in IPA neergeschreven als /ɪk'skjuzɪŋ/.
Onder meer vanwege de moeilijke bestandsencodering van de speciale IPA-tekens gebruikt Nuance een in-house foneemset: L&H+. Dit is een krachtige foneemset die over alle talen reikt en zoals IPA ook klemtoon en lettergreepgrenzen includeert.
Omwille van de leesbaarheid zullen we in deze scriptie gebruik maken van de ARPABET foneemset. Deze foneemset beperkt zich tot alle atomaire klanken uit het Amerikaans Engels, met eventuele toevoeging van klemtoonsuffixen bij de klinkers. Elk foneem uit het ARPABET wordt voorgesteld door een of meerdere ASCII symbolen (integendeel tot de meeste UTF-8 foneemsets), wat manipulatie met scripts vereenvoudigt. Tabel 1 toont het volledige ARPABET met enkele voorbeelden. Het ARPABET wordt ook gebruikt in de publieke CMUDict dataset, welke in een later hoofdstuk aan bod komt.

1.3 Eerder werk

G2P-conversie bestaat reeds een lange tijd. De eerste systemen gebruiken een **rule-based benadering**, waarbij a priori een lexicon met manueel geconverteerde transcripties wordt opgesteld. Wanneer een woord niet in het lexicon gevonden wordt, worden een reeks taalafhankelijke grafeem-naar-foneemregels gebruikt. Deze worden vaak opgesplitst in kleinere submodules per deelprobleem (Elovitz et al., 1976). Een voorbeeld van dergelijke techniek is morfologische decompositie: het proberen splitsen van onbekende woorden in kleinere gekende woorden. Het ontwikkelen van dergelijke regels vereist linguïstische kennis en bovendien hebben sommige talen (zoals Chinees en Japans) een complex schrift wat het moeilijk maakt om alle mogelijke situaties af te handelen. Door propagatie doorheen die submodules gaat soms contextinformatie verloren en wordt er een foute foneemsequentie toegekend. Algemeen geven deze systemen een goede performantie voor gekende woorden, maar falen ze snel bij *out-of-vocabulary* woorden.

Als alternatief op de rule-based benadering worden **joint-sequence modellen** voorgesteld (Chen, 2003; Bisani and Ney, 2008; Galescu and Allen, 2002). Hierbij wordt eerst een grafeem-

Tabel 1: Het ARPABET gebruikt in de CMUDict dataset.

Medelinkers

Foneem	Voorbeeld	Vertaling
B	be	B IY
CH	cheese	CH IY Z
D	dee	D IY
DH	thee	DH IY
F	fee	F IY
G	green	G R IY N
HH	he	HH IY
JH	gee	JH IY
K	key	K IY
L	lee	L IY
M	me	M IY
N	knee	N IY
NG	ping	P IH NG
P	pee	P IY
R	read	R IY D
S	sea	S IY
SH	she	SH IY
T	tea	T IY
TH	theta	TH EY T AH
V	vee	V IY
W	we	W IY
Y	yield	Y IY L D
Z	zee	Z IY
ZH	seizure	S IY ZH ER

Klinkers

Foneem	Voorbeeld	Vertaling
AA	odd	AA D
AE	at	AE T
AH	hut	HH AH T
AO	ought	AO T
AW	cow	K AW
AY	hide	HH AY D
EH	Ed	EH D
ER	hurt	HH ER T
EY	ate	EY T
IH	it	IH T
IY	eat	IY T
OW	oat	OW T
OY	toy	T OY
UH	hood	HH UH D
UW	two	T UW

Klemtoon

Suffix	Betekenis
0	Geen klemtoon
1	Primaire klemtoon
2	Secundaire klemtoon

foneemalignering opgesteld. Een alignering koppelt elk grafeem uit een woord aan één foneem in zijn transcriptie. Op basis van die alignering wordt een verzameling grafeem-foneemparen (zogenaamde *grafonen*) verkregen. Hiermee kan een n-gram model opgesteld worden, dat de probabiliteit van een grafoon gegeven de n vorige grafonen modelleert. Joint-sequence modellen vereisen echter een lexicon waarin de alignering van grafemen en fonemen in aangeduid is, wat extra linguïstische kennis vraagt.

Omdat de alignering in principe een latente variabele is – een middel tot een doel in plaats van het doel zelf – is het interessant om te grafeemsequenties te converteren zonder expliciete alignering. Dankzij de groeiende hoeveelheid data en de stijgende rekenkracht van hedendaagse GPU's is het mogelijk om een **artificieel neuraal netwerk** de grafeem-naar-foneemconversie te laten leren aan de hand van voorbeelden zonder alignering. Eerder onderzoek toont aan dat het gebruik van neurale netwerken groot potentieel heeft (Rao et al., 2015; Bisani and Ney, 2008; Yao and Zweig, 2015; Toshniwal and Livescu, 2016; Yolchuyeva et al., 2019), maar rule-based benaderingen blijven een betere nauwkeurigheid halen. Doordat de neurale netwerken de linguïstische regels zelf leren, hoeven ze niet ingebracht te worden door experts en kan het model sneller in gebruik genomen worden.

Wanneer we nog breder kijken binnen het TTS-domein, zien we dat de G2P-stap in zijn geheel slechts een tussenstap is naar het uiteindelijke doel: tekst naar een spraakgolf converteren. In de literatuur zijn verschillende voorstellen gedaan om de G2P-module, de synthesizer en de normalisatie in één neuraal netwerk te behandelen en rechtstreeks een spraakgolf te simuleren gegeven een grafeemsequentie (Wang et al., 2016; Sotelo et al., 2017; Shen et al., 2018). State-of-the-art architecturen zijn Tacotron van Google (Wang et al., 2017) en Deep Voice van Baidu (Ping et al., 2017). Deze **end-to-end TTS-systemen** hebben het voordeel dat ze geen gebruik hoeven te maken van de relatief beperkte foneemrepresentatie. Ze leren een *impliciet* G2P-model, waardoor prosodie en emotie beter gemodelleerd kunnen worden. Dit voordeel brengt wel met zich mee dat ze moeilijker bij te sturen en te analyseren zijn aangezien de interne representatie niet leesbaar is. Een tweede beperking is het gebrek aan goede data. Een end-to-endsysteem vereist een grote hoeveelheid opgenomen geluidsfragmenten van een professionele spreker, wat veel tijd en geld vraagt.

1.4 Gelijkenissen met machine translation

Machine Translation (MT) is een ander subdomein uit NLP, waarbij een systeem de taak krijgt om een tekst te vertalen van een brontaal naar een doeltaal. De invoer van een MT-model is een sequentie van woorden uit een eindige verzameling van de brontaal (vb. een woordenboek), de uitvoer is een sequentie woorden uit het woordenboek van de doeltaal. De vertaling is uiteraard geen rechtstreekse één-op-één mapping: een woord uit de brontekst kan overeenkomen met één of meerdere woorden uit de doeltekst, die bovendien niet in dezelfde volgorde hoeven te staan. Daarbovenop worden bronwoorden niet consistent naar dezelfde doelwoorden vertaald, maar zijn ze sterk afhankelijk van de context.
Dat G2P sterke gelijkenissen toont met MT wordt snel duidelijk. In plaats van een sequentie woorden hebben we te maken met een sequentie karakters. In plaats van een woordenboek hebben we te maken met een alfabet. De vertaling is niet één-op-één en is sterk afhankelijk van de context.

1.5 Doelstelling

Het doel van deze thesis is het potentieel van neurale netwerken te onderzoeken in het kader van grafeem-naar-foneemconversie op woordniveau. Er wordt onderzocht welke types neurale netwerken geschikt zijn voor deze taak, hun theoretische werking wordt nader bestudeerd en er wordt geëxperimenteerd met verschillende parameters. De resultaten worden zowel onderling als met de het huidige systeem bij Nuance vergeleken, dit op vlak van het aantal correct voorspelde woorden, maar ook op het vlak van geheugengrootte en conversiesnelheid. Verder exploreren we het probleem van grafeem-naar-foneemconversie door de modellen in te schakelen op zinsniveau, met als doel homografen en andere linguïstische effecten beter af te handelen.

1.6 Opbouw van de scriptie

Om de resultaten van de thesis volledig te begrijpen, is er grondige kennis van artificiële neurale netwerken nodig. In Hoofdstuk 2 en 3 bespreken we de basisconcepten van neurale netwerken zo veel mogelijk met betrekking tot G2P.
Vervolgens kunnen we in Hoofdstuk 4 reeds enkele neurale modellen voorstellen en bekijken waarom er nood is aan complexere architecturen.
In Hoofdstuk 5 en 6 komen de belangrijkste principes aan bod: de encoder-decoderarchitectuur en het attentionmechanisme, waarop we variaties zien in Hoofdstuk 7 en 8.
Vervolgens wordt de opstelling van de uitgevoerde experimenten in detail besproken in Hoofdstuk 9, waarna de resultaten geanalyseerd worden in Hoofdstuk 10. Tot slot wordt alles samengevat in Hoofdstuk 11.

2 Introductie tot artificiële neurale netwerken

Vanuit wiskundig perspectief is grafeem-naar-foneemconversie gelijk aan het zoeken naar de uitvoersequentie $\mathbf{y} = (y_1, y_2, \ldots, y_{T_y})$ gegeven een invoersequentie $\mathbf{x} = (x_1, x_2, \ldots, x_{T_x})$. Deze relatie kan gedefinieerd worden door de vectorfunctie $\mathbf{f}$[1]:

$$\mathbf{y} = \mathbf{f}(\mathbf{x}).$$

Deze functie is zeer complex en het opstellen ervan is praktisch onhaalbaar. We kunnen wel een benadering opstellen waarbij het model niet de juiste sequentie teruggeeft, maar wel met een bepaalde waarschijnlijkheid kan zeggen dat een hypothese $\hat{\mathbf{y}}$ de juiste is. Een dergelijk statistisch systeem modelleert een probabiliteitsdistributie over alle mogelijke sequenties aan de hand van zijn parameters θ. Uit die distributie zoeken we dan de sequentie die de grootste waarschijnlijkheid heeft:

$$\begin{aligned}\hat{\mathbf{y}} &= \mathbf{f}(\mathbf{x}, \theta) \\ &= \operatorname*{argmax}_{\mathbf{y}} P(\mathbf{y} \mid \mathbf{x}, \theta).\end{aligned}$$

De parameters θ kunnen geleerd worden op basis van voorbeelden van grafeemsequenties met hun bijhorende transcriptie.
Op basis van deze definitie kunnen we het G2P-probleem opdelen in drie subproblemen (Neubig, 2017):

Modellering – Wat zijn de parameters θ die het model definiëren? Hoeveel parameters zijn er en hoe zijn ze verbonden aan de berekening van de probabiliteitsdistributie?

Leren – Hoe kan het model de parameters θ leren op een efficiënte manier op basis van een eindig aantal voorbeelden?

Zoeken – Hoe zoeken we in de probabiliteitsdistributie de beste hypothese? Dit komt neer op zoeken naar de argmax van de distributie, maar de hoeveelheid mogelijke sequenties stijgt exponentieel met de lengte van de sequentie. Er zal vaak een heuristiek gebruikt worden, zodat niet alle mogelijkheden moeten overlopen worden. De stap die zoekt naar de meest waarschijnlijke sequentie, gegeven een distributie, heet het *inference*-proces.

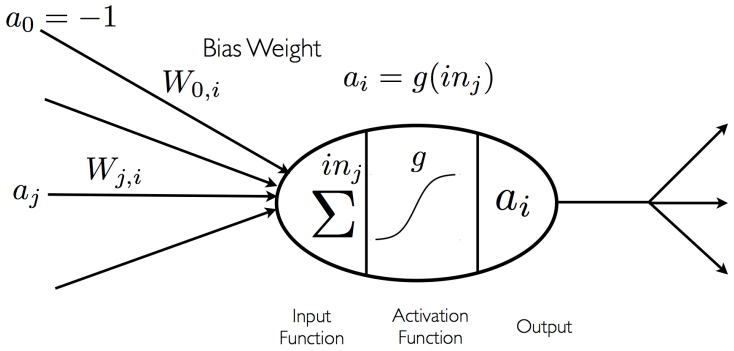
In dit hoofdstuk zullen we de modellering van dergelijk statistisch model met behulp van artificiële neurale netwerken bespreken. Vervolgens bekijken we hoe neurale netwerken kunnen bijleren en hiermee hun nauwkeurigheid optimaliseren. Hierbij worden verschillende concepten geïntroduceerd, zoals loss, gradient descent en minibatching. Het zoeken naar de meest waarschijnlijke sequentie zal in een later hoofdstuk behandeld worden.

2.1 Modellering

Een manier om probabilistische modellen op te bouwen, is via **artificiële neurale netwerken**. Naar analogie met biologische neurale netwerken zijn ze opgebouwd uit sterk verbonden elementaire eenheden, neuronen genaamd (Figuur 2). Neurale netwerken kunnen voorgesteld worden

[1] In deze scriptie worden vectoren steeds vetgedrukt en matrices met een hoofdletter geschreven.

als gerichte, vaak acyclische grafen (*directed acyclic graphs*, DAGs) waarbij de vertices neuronen voorstellen en de verbindingen de invloed voorstellen die neuronen op andere neuronen hebben.



Figuur 2: Mathematische voorstelling van een neuron. De invoer in_j is een combinatie van de activaties van de voorafgaande neuronen, de uitvoer a_i wordt gepropageerd naar de volgende neuronen (Russell and Norvig, 2016).

Neuronen kennen een bepaalde **activatie** a_j (zoals elektrochemische activiteit), die gepropageerd wordt naar de verbonden neuronen en vervolgens hun activatie stimuleert. De gewichten van de verbindingen $w_{i,j}$ bepalen in welke mate de activatie wordt doorgegeven aan de verbonden knopen. De activatie van een neuron j wordt bepaald door de activaties van de voorafgaande knopen:

$$in_j = \sum_{i=1}^{n} (w_{i,j} a_i) + b_j,$$

waarbij b_j een *bias*-term is, die vaak wordt vervangen door $w_{0,j}$ met een een fictieve a_0 input die altijd 1 (of -1) is. Na het combineren van de inputs, wordt een **activatiefunctie** g toegepast om de activatie van neuron j te verkrijgen:

$$a_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j} a_i).$$

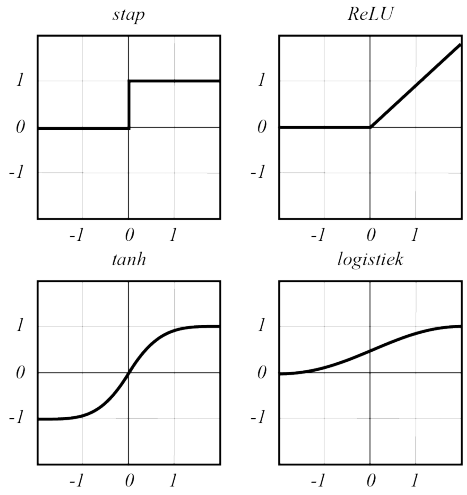
De activatiefunctie introduceert een niet-lineair verband, zodat het gehele netwerk niet-lineaire functies kan representeren. Veelgebruikte activatiefuncties zijn (1) de stapfunctie, (2) de *Rectified Linear Unit* (ReLU), (3) de tangens hyperbolicus en (4) de Sigmoid logistieke functie, allen weergegeven in Figuur 3. Per definitie spreekt men bij neuronen die een activatiefunctie gebruiken van **perceptrons**.

$$\text{Stap}(x) = \begin{cases} 0 & \text{voor } x < 0 \\ 1 & \text{voor } x \geq 0 \end{cases} \tag{1}$$

$$\text{ReLU}(x) = \begin{cases} 0 & \text{voor } x < 0 \\ x & \text{voor } x \geq 0 \end{cases} \tag{2}$$

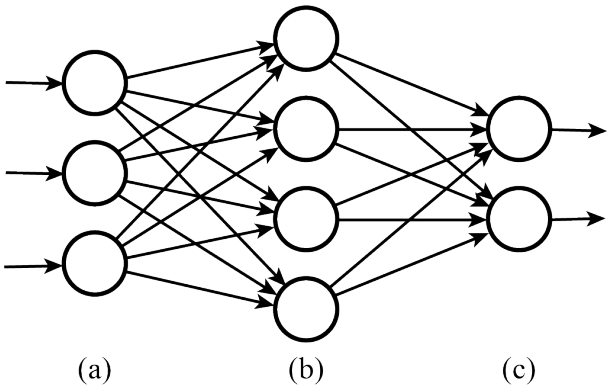
$$\tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})} \tag{3}$$

$$\text{Logistiek}(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{4}$$



Figuur 3: Activatiefuncties.

Neurale netwerken worden vaak in lagen opgebouwd, waarbij elke laag een aantal perceptrons heeft, die elk verbonden zijn met alle perceptrons uit de volgende laag. De eerste en laatste laag worden respectievelijk de in- en uitvoerlagen genoemd, de andere heten *hidden* (verborgen) lagen, omdat de eindgebruiker de activaties van de perceptrons in die lagen niet te zien krijgt. Het geheel van deze verbonden lagen heet een **feedforward neuraal netwerk** of een **multi-layer perceptron** (MLP), weergegeven in Figuur 4. (Russell and Norvig, 2016)



Figuur 4: Een feedforward neuraal netwerk met (a) een invoerlaag van 3 cellen (b) één verborgen laag van grootte 4 (c) een uitvoerlaag van twee cellen.

De kracht van neurale netwerken ligt in het aan elkaar koppelen van een aantal eenvoudige functies om complexere functies weer te geven op een eenvoudig trainbare, parameter-efficiënte manier. In theorie is een multi-layer perceptron met één hidden laag een **universal function-approximator**, wat betekent dat dit model elke mogelijke functie tot een bepaalde nauwkeurigheid kan benaderen, zolang de verborgen laag groot genoeg is. (Neubig, 2017)

Artificiële neurale netwerken werden voor het eerst geïntroduceerd in 1942 (Pitts, 1942). Pas dankzij de stijgende efficiëntie en steeds krachtiger wordende parallelle rekenkracht van GPU's en andere hardware uit de jaren '90 werd het potentieel van neurale netwerken ondervonden.

Merk op dat artificiële neurale netwerken twee use-cases hebben: (1) om meer inzicht te krijgen in biologische neurale netwerken en (2) om specifieke problemen in het domein van artificiële intelligentie op te lossen. In het laatste geval is het dus niet per se de bedoeling een echt biologisch systeem te beschrijven, maar werd de biologie als inspiratiebron gebruikt om artificiële regressie- en classificatiemodellen te verbeteren. Latere neurale netwerken in dit hoofdstuk en verdere hoofdstukken zullen steeds verder afwijken van het biologisch model om de specifieke noden beter te vervullen.
Verder zullen we ons in deze paper enkel toespitsen op ***multiclass* discrete classificatiemodellen**: deze groep modellen heeft als taak een invoer van eender welke vorm (afbeelding, tekst, een aantal meetwaarden...) om te zetten naar één enkel label, gekozen uit een vooraf gedefinieerde (eindige) verzameling labels. Intern zal de laatste laag van een dergelijk model een probabiliteitsdistributie over alle labels teruggeven, waarna het *inference*-proces één label kiest. Later zullen we bij RNN's zien dat G2P-modellen van dit type stap per stap een probabiliteitsdistributie over het foneemalfabet zullen genereren, waarna het *inference*-proces de meest waarschijnlijke sequentie fonemen zoekt, gebruik makend van de distributies over alle stappen heen.

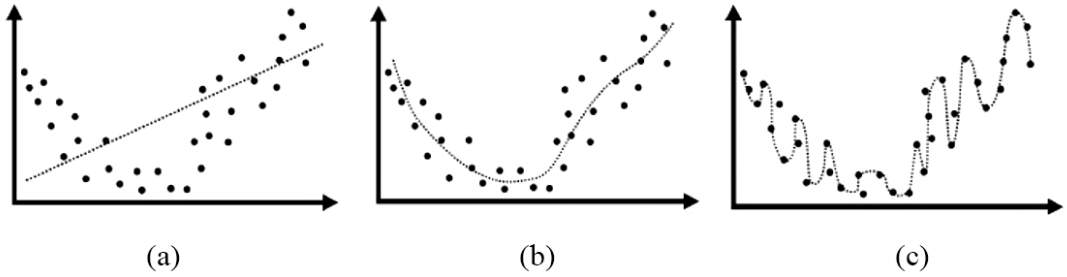
2.2 Leren

In de vorige sectie hebben we een model opgebouwd dat de vectorfunctie $\mathbf{f}(\mathbf{x}) = \mathbf{y}$ met zekere nauwkeurigheid kan benaderen, maar er ontbreekt nog een cruciaal aspect. Initieel kennen we de gewichten $w_{i,j}$ die het neuraal netwerk typeren niet. De bedoeling is dat het netwerk de gewichten zelf zoekt op basis van voorbeelden. In de eerste stap van de training worden iteratief voorbeelden in het netwerk ingegeven om een hypothese te bekomen (*forward calculation*). In het begin zijn deze gewichten met random waarden geïnitialiseerd en is de kans op een correcte hypothese zeer klein. De voorspelling wordt vergeleken met de verwachte uitvoer en er wordt een foutscore berekend. In een tweede fase (*back propagation*) worden de gewichten $w_{i,j}$ aangepast op basis van die foutscore: hoe groter de fout, hoe groter de aanpassing zal zijn.
Terwijl het model meer en meer voorbeelden te zien krijgt zal het zijn gewichten beter kunnen afstemmen zodat het bij alle voorbeelden een zo klein mogelijke fout krijgt. Zo zijn neurale netwerken zeer krachtig als universele functie-approximator, maar deze eigenschap komt wel met enkele tekortkomingen:

- De benaderingsfunctie is **gelimiteerd door de topologie** van het netwerk. Hoe meer neuronen, hoe groter de nauwkeurigheid. Dit heeft wel als gevolg dat er meer parameters zijn die getraind moeten worden, waardoor zowel de *forward* als de *backward pass* trager verlopen.
- De benaderingsfunctie is **gelimiteerd door de voorbeelden** die het voorgeschoteld kreeg. Indien het model alle mogelijke gevallen ziet, zal de benadering hier ook op afgestemd zijn. Vaak beschikken we echter niet over álle mogelijke gevallen, want een taal is niet eindig. Bovendien is er geen nood aan een neuraal netwerk wanneer we voor alle mogelijke woorden de transcriptie kennen. Kortom, het model moet in staat zijn om te **generaliseren** op basis van een beperkt aantal voorbeelden. Indien een netwerk te veel voorbeelden ziet, zal het zich te veel fixeren op die beperkte set en zal er **overfitting** optreden. Figuur 5 (c) toont hoe een netwerk kan overfitten.

Overfitting treedt ook op wanneer een netwerk te veel neuronen heeft. Langs de andere kant kan

een netwerk ook *underfitten* (Figuur 5 (a)) wanneer het te weinig neuronen heeft of te weinig voorbeelden zag. Een goed opgesteld en goed getraind netwerk zal goed kunnen generaliseren, zoals weergegeven in Figuur 5 (b).



Figuur 5: Het resultaat van een benaderingsmodel met (a) underfitting (b) een goede fitting en (c) overfitting.

2.2.1 Trainings-, test- en validatiesets

De dataset wordt dikwijls in drie partities verdeeld:

Trainingsdata – wordt gebruikt om de gewichten van het model aan te passen.

Testdata – is een compleet onafhankelijke verzameling samples die het model nog nooit gezien heeft. Er zijn geen keuzes tussen verschillende modellen gemaakt op basis van deze data. Ze biedt zo een manier om de werkelijke nauwkeurigheid te meten, vergelijken en publiceren.

Validatiedata – wordt gebruikt om te kiezen tussen verschillende modellen, om parameters aan te passen en om overfitting te voorkomen. Als de validatieset een goede steekproef is van de trainingsset, kunnen we overfitting voorkomen door de trainingsfase te stoppen wanneer de nauwkeurigheid op de validatieset niet meer stijgt. Men spreekt ook soms ook van een *development* set.

Typische ratios zijn (80%, 10%, 10%) en (60%, 20%, 20%). Het is belangrijk dat de validatieset en de testset goede representaties zijn voor de trainingsset. Daarom moeten de gegevens eerst random gepermuteerd worden, zodat ze bijvoorbeeld zeker niet in alfabetische volgorde gesplitst worden. Een andere reden voor deze *random shuffle* is dat bij technieken zoals online learning en minibatching (zie verder) de performantie van het model beïnvloed wordt door de samples die het vlak ervoor te zien kreeg.

2.2.2 Loss

Vooraleer we de gewichten kunnen updaten, moeten we een metriek vastleggen die de prestaties van het model bepaalt. Zo weten we of het model op de goede weg is of als de wijzigingen het model alleen maar slechter maken. Zoals eerder aangehaald, zal een G2P-model per te voorspellen foneem een probabiliteitsdistributie over het foneemalfabet teruggeven. Die voorspelde distributie $\hat{P}(\mathbf{y} \mid \theta)$ vergelijken we met de verwachte distributie $P(\mathbf{y})$, die overal nul is behalve bij het verwachte foneem. Op die manier kunnen we een **loss-functie** of cost-functie vastleggen;

een score die ons vertelt hoe slecht het model presteert bij de voorspelling van een foneem. Er zijn verschillende opties voor deze functie, zoals (5) de absolute afwijking L_1 en (6) de kwadratische afwijking L_2. In het geval van een multiclass discrete classificatietaak waarbij de klassen elkaar uitsluiten (er is slechts één label mogelijk), blijkt (7) de **cross-entropy loss** L_{CE} een goede keuze. Ze werken alle drie door per symbool uit het uitvoeralfabet (met grootte V) een foutwaarde te berekenen en al die foutwaarden te sommeren.

$$L_1(P, \hat{P}) = \sum_{i=1}^{V} |P(y_i) - \hat{P}(y_i \mid \theta)| \tag{5}$$

$$L_2(P, \hat{P}) = \sum_{i=1}^{V} (P(y_i) - \hat{P}(y_i \mid \theta))^2 \tag{6}$$

$$L_{CE}(P, \hat{P}) = -\sum_{i=1}^{V} P(y_i) \log(\hat{P}(y_i \mid \theta)) \tag{7}$$

2.2.3 Gradient descent

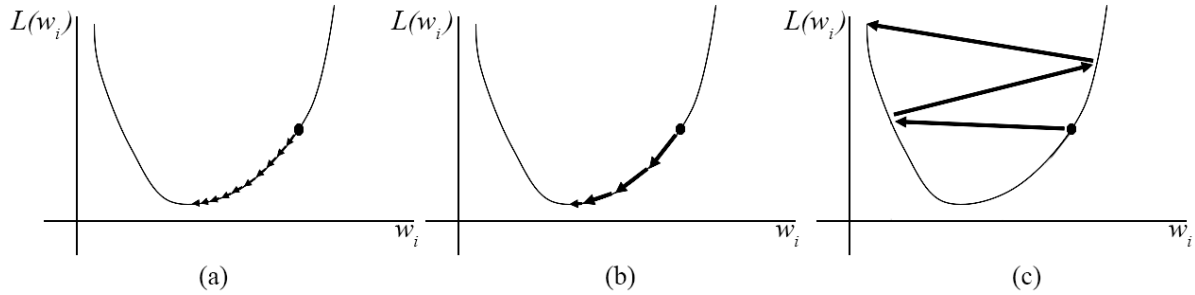
We hebben een metriek vastgelegd die ons inzicht geeft over de prestaties van een model op een uitvoersymbool. Het doel is nu om die loss zo klein mogelijk te maken door de gewichten van het model aan te passen. Dit doen we door de partiële afgeleide voor alle gewichten w_i gelijk te stellen aan nul.

$$\nabla \text{Loss}(\mathbf{w}) = \left(\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \ldots, \frac{\partial}{\partial w_n} \right) \text{Loss}(\mathbf{w}) = \mathbf{0}$$

In vele gevallen bestaat er geen eindige oplossing voor dit stelsel of is het zoeken ernaar niet haalbaar. We kunnen wel de gewichten iteratief aanpassen zodanig dat de loss daalt. Als de partiële afgeleidde $\frac{\partial}{\partial w_i}\text{Loss}(\mathbf{w})$ positief is, maken we het gewicht w_i kleiner. Is ze negatief, dan maken we het gewicht groter. De hoeveelheid van dit dalen/stijgen is bepaald door de grootte van de afgeleide en een constante factor α, via:

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} \text{Loss}(\mathbf{w}), \forall i.$$

De factor α is de **learning rate**. Ze kan een vaste constante zijn of kan decrementeren naarmate het model beter getraind is. Een hogere learning rate versnelt de convergentie, maar indien ze te hoog genomen wordt kan ze resulteren in suboptimale convergentie of zelfs divergentie, zoals weergegeven in Figuur 6. Het proces dat iteratief de gewichten aanpast heet **gradient descent**. We kunnen dit visueel voorstellen als het afdalen van een curve volgens de negatieve gradient. (Russell and Norvig, 2016)



Figuur 6: Gradient descent daalt af volgens de helling van de curve. (a) Een kleine learning rate leidt tot trage convergentie (b) Een goede learning rate maakt gradient descent zowel snel als betrouwbaar. (c) Een te grote learning rate leidt tot divergentie. (Jordan, 2018)

2.3 Batching

De trainingsfase bestaat uit twee stappen: *forward calculation* waarbij de loss berekend wordt en *back propagation* waarbij de gewichten via gradient descent gewijzigd worden. Die twee stappen worden meerdere keren op elke sample in de trainingsdataset toegepast, tot de loss niet meer verminderd kan worden. Wanneer we de hele dataset één keer overlopen hebben, is een **epoch** verstreken.

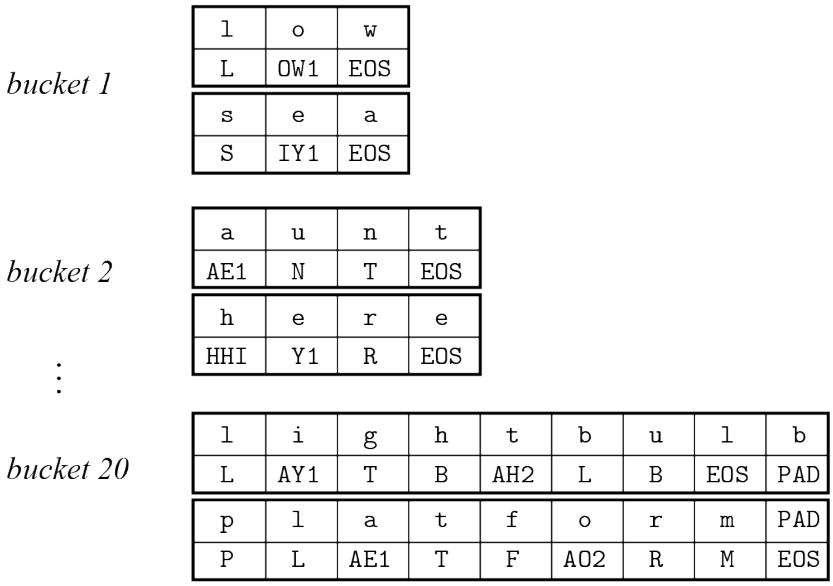
De techniek waarbij we voor elke sample in de trainingsset de loss berekenen en de gewichten updaten, heet **online learning**. Daartegenover staat **batch learning**, waarbij we de gehele trainingsset als een geheel beschouwen en op het einde het gemiddelde van de gradiënten berekenen en toepassen. Beide strategieën hebben hun voor- en nadelen:

- Online learning kan snel een aanvaardbaar model verkrijgen, zonder de volledige trainingsset te hoeven doorlopen. Batch learning daarentegen moet de gehele dataset doorlopen vooraleer het een eerste update kan doorvoeren.
- Online learning wordt beïnvloed door de volgorde van de trainingssamples, waardoor het in verhouding met batch learning tot een minder stabiel resultaat leidt.
- Omwille van die random factor (waarbij de volgorde invloed heeft), heeft online learning minder kans om in een lokaal optimum vast te zitten in vergelijking met batch learning.

Minibatching is een goede trade-off tussen online en batch learning. Hierbij nemen we groepjes van n samples samen in een minibatch en wijzigen we de gewichten na de forward pass van elke sample in de minibatch. Typische batch-sizes zijn $n = 64$ en $n = 128$. Als de batch-size kleiner is, zal het model zijn gewichten nauwkeuriger updaten, maar zal elke epoch langer duren. Doordat GPU's zeer goed met matrices overweg kunnen, zullen de activaties voor de samples in een batch ook effectief parallel berekend worden.

Bij veel toepassingen is de lengte van de invoersequentie altijd even groot. Bij G2P en MT is dit echter niet het geval. Wanneer minibatching gebruikt wordt, moeten alle samples in een batch wel even groot zijn om de parallellisatie te benutten. Dit wordt opgelost door speciale padding in de vorm van `<pad>`-tekens toe te voegen aan de kortere sequenties. De trainingsfase herkent deze tekens en weet dat ze niet meetellen om o.a. de loss te berekenen.

Een extra optimalisatie zal sequenties van ongeveer dezelfde lengte in dezelfde minibatch groeperen, waardoor niet te veel computatie verloren gaat aan de padding. Dit mechanisme heet **bucketing**, omdat we k buckets van verschillende lengte maken. Figuur 7 toont minibatching van $n = 2$ met bucketing van $k = 20$ buckets. (Neubig, 2017)



Figuur 7: Minibatching met een batch-size $n = 2$ en een bucketing van $k = 20$ buckets. ($n = 2$ is geen realistische waarde. In de praktijk zal $n = 128$ gebruikt worden.)

3 Recurrent Neural Networks

Feedforward neurale netwerken kennen zeer veel toepassingen in classificatie- en regressieproblemen. De input is een vast aantal features, zoals de lengte van een plant, de grootte en de kleur van zijn bladeren. De features kunnen ook abstracter zijn, zoals pixelwaarden uit een afbeelding van een handgeschreven getal. Het model moet een label voorspellen, zoals het type plant of het getal dat herkend wordt. Elke voorspelling is onafhankelijk van de vorige voorspelling en het aantal ingevoerde features is altijd gelijk. Bij G2P daarentegen hebben we te maken met woorden van verschillende lengten. We moeten dus een manier vinden om met feedforward neurale netwerken een woord van variabele lengte om te zetten in een nieuw woord met een andere lengte. De letters één voor één invoeren is geen optie, want ze zijn afhankelijk van de vorige ingevoerde en de volgende in te voeren letter.

Recurrent Neural Networks (RNN's) is een verzamelnaam voor neurale netwerken die een interne toestand bewaren doorheen een sequentie. RNN's kunnen hun interne toestand (geheugen) gebruiken om reeksen inputs te verwerken en context te bewaren. De term RNN kan zowel naar de overkoepelende vorm verwijzen als naar de eenvoudigste vorm (Elman, 1990) welke we verder benoemen als een **Elman RNN**.

3.1 Encodering

Een neuraal netwerk verwacht voor elke invoercel een reële startwaarde, de inputvector. In het G2P-probleem vertrekken we van een woord. Er zijn enkele manieren om woorden om te zetten in reële vectoren:

Ordinaal – Een naïeve oplossing zou aan elke letter van het alfabet een unieke identificatie toekennen. Als we de positie van een letter in het alfabet als ID gebruiken, kan het woord "cat" op volgende manier geëncodeerd worden:

$$\text{“}cat\text{”} \mapsto \begin{bmatrix} 2 & 0 & 19 \end{bmatrix}$$

Een neuraal netwerk werkt intern echter niet discreet, maar met reële getallen. In een hidden laag kan de vector $\begin{bmatrix} 1,5 & 3,35 \end{bmatrix}$ dus zeker voorkomen. Dit impliceert dat er een relatie is tussen letters die alfabetisch dicht bijeen liggen en dat er bovendien een ordinale relatie is tussen de letters, namelijk $a < c < t$. Deze relaties zijn in de werkelijkheid niet aanwezig en zullen de prestaties van een model negatief beïnvloeden.

One-hot – Een one-hot encoder maakt voor elke letter een volledige vector in plaats van een enkel getal. Een cel krijgt waarde 1 als die zich op de rij van het unieke ID bevindt, anders krijgt die 0. Een voorbeeld van het woord "cat":

$$\text{“}cat\text{”} \mapsto \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Embedding – Naargelang de grootte van het gebruikte alfabet stijgt, wordt de one-hot geëncodeerde vector onhandelbaar groot. Een trade-off tussen ordinale encodering en one-hot encodering is *character embedding*. Hierbij wordt een letter geëncodeerd naar een unieke n-dimensionale reële vector, door een gewichtenmatrix met dimensies $n \times V$ te vermenigvuldigen met de one-hot encodering (met V de grootte van het alfabet). De vectoren worden zo gekozen dat ze in de n-dimensionale ruimte voldoende ver uit elkaar liggen om onderscheiden te worden. Een voorbeeld voor een 2-dimensionale embedding van het woord "cat":

$$\text{“}cat\text{”} \mapsto \begin{bmatrix}0,53\\0,85\end{bmatrix}, \begin{bmatrix}0,60\\0,80\end{bmatrix}, \begin{bmatrix}-0,78\\-0,62\end{bmatrix}$$

Een voordeel bij embedding dat oorspronkelijk uit *machine translation* komt, is dat woorden met ongeveer dezelfde betekenis in de embeddingruimte dichter bij elkaar kunnen liggen dan woorden met een sterk verschillende betekenis. Hierdoor kan het model beter generaliseren dan een one-hot-encodering, waar vergelijkbare woorden niks delen. De gewichtenmatrix nodig voor de encodering kan mee getraind worden in het netwerk, zodat die zelf leert welke woorden dicht bij elkaar liggen. Als we dit voordeel vertalen naar G2P, betekent dit dat letters met sterk gelijkende betekenis dicht bij elkaar liggen. In het Latijns alfabet komt dit neer op letters en hun hoofdlettervariant, leestekens en getallen. In het fonetisch schrift zullen idealiter klinkers en medeklinkers zichtbare clusters vormen.
Character embedding blijkt vooral nuttig om logografische schriften met een groot alfabet te reduceren tot een kleinere ruimte, zoals Chinees (±50 000 karakters).

3.2 Recurrent Neural Networks

Het Elman RNN (Elman, 1990) kan gezien worden als een feedforward netwerk met één *hidden layer* waarbij samen met de geëncodeerde input ook de **interne toestand** (*hidden state*) van de vorige iteratie wordt meegeven, om vervolgens een uitvoer en een nieuwe interne toestand terug te krijgen.

Voor een geëncodeerde invoersequentie $X = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{T_x})$ wordt de interne toestand $\mathbf{h}_t$ voor (tijds)stap t als volgt berekend:

$$\mathbf{h}_t = g(W\mathbf{x}_t + U\mathbf{h}_{t-1} + \mathbf{b}_h), \tag{8}$$

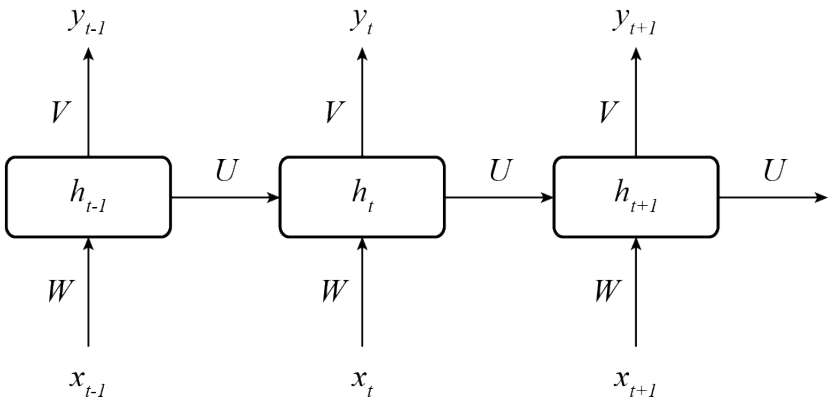
waarbij W, U en $\mathbf{b}_h$ trainbare parameters zijn. Voor de initiële interne toestand $\mathbf{h}_0$ kan een ruisvector gebruikt worden, een constante waarde of ze kan beschouwd worden als een trainbare parameter. Figuur 8 toont een veel voorkomende representatie van een RNN, waarbij de cellen (rechthoeken) telkens (8) voorstellen, maar op een andere tijdsstap. Merk op dat de gewichten W, U en V op de figuur telkens naar dezelfde waarde verwijzen. Ze zijn niet afhankelijk van de stap t.
RNN's worden in hun geheel getraind via *back-propagation through time* (BPTT), een principe analoog aan klassieke backpropagation (Mousa and Schuller, 2016).

Om een uitvoerdistributie $\hat{\mathbf{p}}_t$ te genereren kan rechtstreeks $\mathbf{h}_t$ gebruikt worden of ze wordt berekend door:

$$\hat{\mathbf{p}}_t = \text{softmax}(V \cdot h_t + \mathbf{b}_y)$$

met V en $\mathbf{b}_y$ extra trainbare parameters. Een **softmax**-functie zet een vector van scores (reële getallen) om naar een probabiliteitsdistributie: elk element krijgt een waarde binnen $[0; 1]$ en de totale som is gelijk aan 1:

$$\text{softmax}(\mathbf{y})_i = \frac{\exp(y_i)}{\sum_{j=1}^{V} \exp(y_j)}$$



Figuur 8: Een Elman RNN.

3.3 Long Short-Term Memory Networks

Long short-term memory networks (LSTM) zijn een uitbreiding op Elman RNN's om het **vanishing gradient** probleem te vermijden. Dit is een probleem dat optreedt tijdens de gradient descent waardoor de outputs op latere tijdstippen meer invloed hebben dan de eerdere. Een LSTM (Figuur 9) introduceert een memory cell $\mathbf{c}$ en drie *poorten*: de invoerpoort $\mathbf{i}$, de uitvoerpoort $\mathbf{o}$ en de vergeetpoort $\mathbf{f}$:

$$\mathbf{f}_t = g(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1})$$

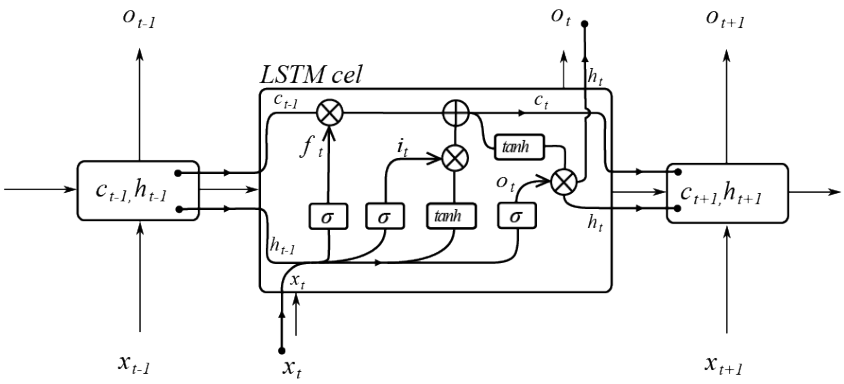
$$\mathbf{i}_t = g(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1})$$

$$\mathbf{o}_t = g(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1})$$

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ g(W_c \mathbf{x}_t + U_c \mathbf{h}_{t-1})$$

$$\mathbf{h}_t = \mathbf{o}_t \circ g(\mathbf{c}_t) = \mathbf{y}_t.$$

Hierbij is de $\circ$-operator een elementsgewijze vermenigvuldiging van vectoren. De invoerpoort $\mathbf{i}_t$ bepaalt de mate waarin de huidige invoer $\mathbf{x}_t$ de interne toestand $\mathbf{c}_t$ beïnvloedt. De vergeetpoort bepaalt in hoeverre een waarde van de interne toestand $\mathbf{c}_{t-1}$ onthouden blijft en de uitvoerpoort bepaalt welke elementen uit de interne toestand $\mathbf{c}_t$ gebruikt worden om de uitvoer $\mathbf{y}_t$ te berekenen. De activatiefunctie g van de LSTM-poorten is vaak de logistieke functie uit (4).



Figuur 9: Een gedetailleerde voorstelling van een LSTM-cel.

3.4 Gestapelde en bidirectionele RNN's

Een belangrijke eigenschap van RNN's is dat ze gestapeld kunnen worden door de uitvoer van de vorige te gebruiken als invoer (**stacked RNNs**). Voor een 3-laags gestapeld RNN zien de berekeningen op stap t er als volgt uit:

$$\mathbf{h}_t^{[1]} = \text{RNN}_1(\mathbf{x}_t, \mathbf{h}_{t-1}^{[1]})$$

$$\mathbf{h}_t^{[2]} = \text{RNN}_2(\mathbf{h}_t^{[1]}, \mathbf{h}_{t-1}^{[2]})$$

$$\mathbf{h}_t^{[3]} = \text{RNN}_3(\mathbf{h}_t^{[2]}, \mathbf{h}_{t-1}^{[3]}),$$

waarbij $\mathbf{h}_t^{[l]}$ de hidden state is voor de l-de laag op tijdstip t en RNN$(\cdot)$ een afkorting is voor de vergelijking die de hidden state berekent, afhankelijk van het type RNN. Meerdere RNN's stapelen kan nuttig zijn om dezelfde reden dat we meer hidden layers toevoegen bij feedforward netwerken: ze zijn beter in het generaliseren van complexere verbanden (Neubig, 2017).

Wanneer er ook informatie nodig is over toekomstige tijdstippen, zoals bij G2P, waar een lezer vooruit kan kijken om extra context te geven, is de **bidirectionele RNN**-architectuur (Bi-RNN) zeer krachtig. In een bidirectionele RNN verwerkt één RNN-laag de input van links naar rechts, terwijl de andere de input van rechts naar links verwerkt. Vervolgens worden de outputs van de twee lagen gecombineerd door ze te concateneren:

$$\overrightarrow{\mathbf{h}_t} = \text{RNN}(\mathbf{x}_t, \overrightarrow{\mathbf{h}_{t-1}})$$

$$\overleftarrow{\mathbf{h}_t} = \text{RNN}(\mathbf{x}_t, \overleftarrow{\mathbf{h}_{t+1}})$$

$$\mathbf{h}_t^{[l]} = [\overrightarrow{\mathbf{h}_t}; \overleftarrow{\mathbf{h}_t}]. \tag{9}$$

Bidirectionele RNN's zijn ook stapelbaar, maar dan moet vanwege de concatenatie in (9) de grootte van de hidden state van elke laag $\mathbf{h}_t^{[l]}$ het dubbel zijn van $\mathbf{h}_t^{[l-1]}$. In dat geval is het eenvoudiger om een extra trainbare hidden layer tussen te voegen zoals

$$\mathbf{h}_t^{[l]} = W_1\overrightarrow{\mathbf{h}_t} + W_2\overleftarrow{\mathbf{h}_t} + \mathbf{b},$$

die beide lagen combineert zonder de dimensies te verdubbelen (Graves et al., 2013).
In NLP worden bidirectionele LSTM's (BLSTM) zeer vaak gebruikt. Voor veel applicaties biedt de toekomstige invoer noodzakelijke context aan de hypothese op de huidige tijdsstap, iets waar klassieke RNN's geen weet van hebben. Het principe werd reeds succesvol toegepast op spraakherkenning en machine translation.

4 G2P-modellen gebaseerd op RNN's

Als we RNN's toepassen op het G2P-probleem, wordt er in elke stap een encodering (one-hot of via character embedding) van een grafeem ingevoerd in de onderste laag van een RNN-stack. Dit resulteert bij elke stap in een probabiliteitsdistributie aan de uitvoer van de RNN-cel. Deze aanpak kent reeds een eerste tekortkoming: grafeem- en foneemsequenties zijn niet noodzakelijk even lang. Het verschil in lengte is te verklaren via alignering. Een alignering probeert één of meerdere grafemen te corresponderen met één of meer fonemen. Zo komt de `SH`-klank (in IPA: /ʃ/) uit "ship" duidelijk overeen met de letters "sh", maar is ze al veel moeilijker te aligneren in het woord "national". Het is ook mogelijk dat een grafeem overeenkomt met twee fonemen, zoals de "x" die vaak gealigneerd wordt met `K S`.

4.1 Modellen met alignering

Een alignering moet aanwezig zijn in de dataset zelf. Dit kan verwezenlijkt worden op verschillende manieren. Een optie is om een speciaal *null*-symbool of ϵ-symbool toe te voegen zodat er een één-op-één alignering kan toegekend worden (Rao et al., 2015). Zo wordt het woord "able", dat uitgesproken wordt als `EY B AH L` in ARPABET-notatie, op de volgende manier gealigneerd:

grafemen	A	B	ϵ	L	E
fonemen	`EY`	`B`	`AH`	`L`	ϵ

Een andere theorie gebruikt many-to-many alignments, meer specifiek {1,2}-to-{0,1,2} alignments, waarbij één of twee grafemen kunnen resulteren in één of twee fonemen, of een speciaal *null*-foneem (Mousa and Schuller, 2016). Het woord "excusing" wordt op de volgende manier gealigneerd, waarbij een dubbelepunt twee symbolen groepeert:

grafemen	E	X	C	U	S	I	N:G
fonemen	`IH`	`K:S`	`K`	`Y:UW`	`Z`	`IH`	`NG`

Er bestaan ook nog steeds gevallen waarbij een grafeem geen overeenkomstig foneem heeft:

grafemen	L	A	T	E
fonemen	`L`	`EY`	`T`	ϵ

In sommige talen is de alignering veel moeilijker om eenduidig vast te leggen. Vaak is er controversie over de effectieve alignering en het opstellen van dergelijke dataset is moeilijk.

4.2 Modellen zonder alignering

Voor datasets zonder alignering is er kennis over de sequentie invoerkarakters en uitvoerkarakters, maar niet over de rechtstreekse relatie ertussen.
Als men het model traint om uit een grafeemsequentie rechtstreeks de foneemsequentie te voorspellen, zou een unidirectioneel LSTM-netwerk verkeerde relaties leren en slechte kwaliteit tot gevolg hebben. Gegeven onderstaand voorbeeld:

grafemen	A	B	L	E
fonemen	EY	B	AH	L

Hier leert het netwerk dat een L-grafeem overeenkomt met een AH-foneem, wanneer het in de sequentie (A, B, L) voorkomt. Dit is echter een foute veralgemening (bijvoorbeeld "ablaze": AH B L EY Z).
Om dit aligneringsprobleem op te lossen wordt het concept van een **output delay** geïntroduceerd (Rao et al., 2015). Hierdoor geven we de LSTM de kans om d aantal grafemen voorop te kijken en een context op te bouwen voordat het een karakter moet teruggeven. Onderstaand voorbeeld hanteert een vaste delay van $d = 2$. Ongelijke grafeem- en foneemsequenties worden afgehandeld door extra padding-karakters <p> in te voeren.

grafemen	E	X	C	U	S	I	N	G	<p>	<p>	<p>
fonemen	<p>	<p>	IH	K	S	K	Y	UW	Z	IH	NG

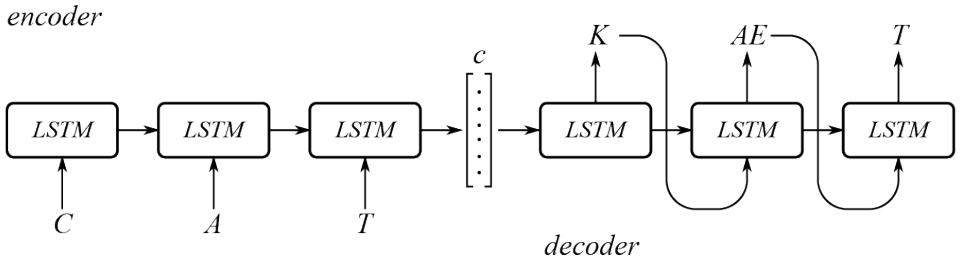
Om het netwerk toegang te geven tot de volledige context kan een full-delay ($d = T_x$) gebruikt worden. Hierbij wordt het model getraind om padding-karakters te voorspellen tot het einde van het woord bereikt is. Dan pas mag het model de fonemen teruggeven (met een invoer van padding-karakters). Vaak wordt er dan ook een end of sequence token <eos> gebruikt om het model extra te begeleiden.

grafemen	A	B	L	E	<eos>	<p>	<p>	<p>
fonemen	<p>	<p>	<p>	<p>	EY	B	AH	L

Voor BLSTM's is er geen probleem aangezien ze de volledige input beschouwen (als combinatie van $\overrightarrow{\mathbf{h}}$ en $\overleftarrow{\mathbf{h}}$) vooraleer er een uitvoer wordt gegenereerd. BLSTM's zijn wel trager en bevatten dubbel zoveel parameters in vergelijking met unidirectionele LSTM's.
Merk op dat het converteren van grafeem- naar foneemsequenties complexer is wanneer de dataset geen alignering bevat en de dimensies van het RNN groter moeten zijn om dezelfde performantie te bekomen.

5 De encoder-decoderarchitectuur

Een encoder-decoderarchitectuur (Cho et al. (2014), Sutskever et al. (2014)) wijkt in zijn eenvoudigste vorm niet veel af van een RNN-model zonder alignering met een *full delay*, besproken in Sectie 4.2. In plaats van de invoersequentie rechtstreeks te vertalen naar een uitvoersequentie, gebruikt de encoder-decoderarchitectuur een encoder RNN die de invoersequentie omzet in een vector met vaste lengte en een decoder RNN die de vector decodeert tot een uitvoersequentie. Figuur 10 toont hoe een encoder-decoder model het woord "cat" eerst omzet naar een vector **c**, die vervolgens door de decoder vertaald wordt naar een foneemsequentie.



Figuur 10: Een encoder LSTM zet *CAT* om in een contextvector c, de decoder decodeert de contextvector naar `K AE T`.

5.1 Encoder

De encoder is een RNN dat de invoersequenties van variabele lengte omzet naar een vector met vaste lengte, de **contextvector c**. Deze contextvector (*thought vector*, *feature vector*) bevat genoeg informatie over de invoersequentie om ze te decoderen. Ze kan geïnterpreteerd worden als de verzameling van alle geziene grafemen, inclusief de context waarin ze voorkomen, maar in de praktijk is ze abstracter dan dat. Een typische keuze voor **c** is de laatste *hidden state* $\mathbf{h}_{T_x}$ van de encoder. Alle uitvoer van de LSTM's wordt verworpen, ervan uitgaande dat die ergens aanwezig zijn in de laatste hidden state. Om dit te verwezenlijken, verkiezen we LSTM's boven eenvoudige RNN's omdat ze een langer geheugen hebben en dus de eerste karakters beter zullen bijhouden in de contextvector.

Stacked en bidirectionele encoders

Uiteraard is er ook de mogelijkheid om LSTM's te stapelen (zie Sectie 3.4) bij de encoder of om gebruik te maken van bidirectionele LSTM's. In dit geval wordt de contextvector opgebouwd uit de laatste hidden states van alle lagen. Ze vereist wel dat de decoder even veel lagen heeft. Hoewel een unidirectioneel model ook weet heeft van de volledige invoersequentie blijken bidirectionele modellen robuuster tegen verschillen in volgorde bij in- en uitvoersequentie.

Reversed encoders

Uit experimenten (Sutskever et al., 2014) blijkt dat de contextvector beter wordt opgesteld wanneer de grafeemsequentie in omgekeerde volgorde wordt ingevoerd ("`c a t <eos>`" wordt

"`t a c <eos>`"). Wanneer de inputsequentie in normale volgorde in de LSTM wordt ingevoerd, bevinden de afhankelijkheden tussen invoer- en uitvoersymbolen zich ongeveer T_x stappen uit elkaar (uitgaande van een stabiele alignering). Het *vanishing gradient*-probleem zorgt ervoor dat afhankelijkheden moeilijker te onthouden zijn voor de LSTM naarmate ze verder uiteen liggen. Wanneer we de sequentie omkeren, worden de laatst ingevoerde karakters bijna onmiddellijk vertaald in hun fonemen. Op die manier maken we het voor het model eenvoudiger om vroeg in het trainingsproces reeds juiste voorspellingen te maken. Later kan het zich focussen op de verder uiteen liggende afhankelijkheden.

5.2 Decoder

De decoder gebruikt de contextvector $\mathbf{c}$ om stap voor stap een probabiliteitsdistributie $\hat{\mathbf{p}}_t$ te genereren voor het symbool y_t op positie t in de foneemsequentie. De decoder is ook een RNN waarvoor vaak een LSTM gekozen wordt. De LSTM-cellen kunnen ook gestapeld worden, maar niet bidirectioneel. De hidden state van de eerste tijdsstap wordt geïnitialiseerd op $\mathbf{c}$ en voor de eerste invoer $\mathbf{y}_0$ gebruiken we een speciaal *start of sequence* `<sos>`-token. Voor de invoersequentie van de decoder zijn er enkele opties:

Output-as-input – De invoer op stap t is de uitvoer van de decoder op de vorige stap $t-1$, namelijk de probabiliteitsdistributie over het foneemalfabet. Dit is de meest eenvoudige implementatie, maar leidt tot langzame convergentie, een instabiel model en dus slechte performantie.

Teacher forcing – De invoer op stap t is de verwachte uitvoer van de decoder op stap $t-1$, met name de *ground truth* of *targetsequentie*. Hierdoor convergeert het model sneller, omdat het bij het begin van het trainingsproces niet beïnvloed wordt door de willekeurige output uit de vorige tijdsstap van het nog niet geconvergeerde model. Helaas beschikt men enkel over de ground truth tijdens het trainingsproces (op gelabelde data). Bij effectief gebruik van het model (*inference* op ongeziene data) moet men bij teacher forcing een andere input aan de decoder meegeven. Hiervoor gebruikt men ofwel de probabiliteitsdistributie $\hat{\mathbf{p}}_{t-1}$ die het model op de vorige tijdsstap produceerde, ofwel de effectieve voorspelling van de decoder $\hat{\mathbf{y}}_{t-1}$. Wanneer het model echter een fout symbool voorspelt, zal die fout zich accumuleren doorheen de volledige sequentie.

Scheduled sampling – Scheduled sampling lost het probleem bij *inference* grotendeels op door tijdens het trainingsproces bij elke stap met probabiliteit ϵ te kiezen tussen de *ground truth* uit de vorige stap $\mathbf{y}_{t-1}$, of de output $\hat{\mathbf{p}}_{t-1}$ gegenereerd door het model (met kans $\epsilon - 1$). De waarde voor ϵ start bij 1 en daalt (lineair, exponentieel of via een inverse sigmoid) tot 0, naarmate het model convergeert. Hierdoor convergeert het model sneller dan bij *output-as-input* én is het robuuster tegen fouten dan *teacher forcing*. (Bengio et al., 2015)

5.3 Uitvoer genereren: greedy search en beam search

De decoder geeft geen waarde terug, maar een reeks *logits* (scores voor waarschijnlijkheid) over alle mogelijke fonemen. De logits worden door een softmaxlaag omgezet naar een kans, i.e. de kans dat dat foneem het verwachte foneem is. Voor een klassiek classificatieprobleem geeft de argmax de beste uitvoerwaarde, maar voor een sequence-to-sequenceprobleem kan deze *greedy*

strategie tot een suboptimaal resultaat leiden.
Stel dat we een invoersequentie "AB" moeten omzetten naar een uitvoersequentie "BA", waarbij het invoer- en uitvoeralfabet beide bestaan uit $\{A, B\}$. De (fictieve) probabiliteiten van het model zijn:

$$\hat{P}(\mathbf{y}_1 \mid \mathbf{x}) = \begin{matrix} \texttt{A} & \texttt{B} \\ \left[\ 0,60 \quad 0,40\ \right] \end{matrix}$$

Een **greedy strategie** kiest voor $y_1 = \texttt{A}$ als eerste karakter en bepaalt de volgende probabiliteit:

$$\hat{P}(\mathbf{y}_2 \mid \texttt{A}, \mathbf{x}) = \begin{matrix} \texttt{A} & \texttt{B} \\ \left[\ 0,45 \quad 0,55\ \right] \end{matrix}$$

Na het kiezen van het meest waarschijnlijke karakter $y_2 = \texttt{B}$ krijgen we een totale waarschijnlijkheid van:

$$\begin{aligned} \hat{P}(\texttt{AB} \mid \mathbf{x}) &= \hat{P}(\texttt{A} \mid \mathbf{x}) \cdot \hat{P}(\texttt{B} \mid \texttt{A}, \mathbf{x}) \\ &= 0,33 \end{aligned}$$

Wanneer we echter voor y_1 het andere karakter kiezen, verkrijgen we (fictief):

$$\hat{P}(\mathbf{y}_2 \mid \texttt{B}, \mathbf{x}) = \begin{matrix} \texttt{A} & \texttt{B} \\ \left[\ 0,90 \quad 0,10\ \right] \end{matrix}$$

$$\hat{P}(\texttt{BA} \mid \mathbf{x}) = 0,36$$

Het is dus duidelijk dat een betere inference-strategie gezocht kan worden. Het exacte maximum kan enkel gevonden worden door alle mogelijke sequenties te bekijken en

$$\operatorname{argmax} \prod_{t=1}^{T_y} \hat{P}(y_t \mid \mathbf{x}, y_1, \ldots, y_{t-1})$$

te berekenen. Het aantal mogelijke sequenties is helaas te veel om efficiënt te bekijken, dus daarom stelt men een heuristiek voor, genaamd **beam search**.

Bij *beam search* houdt men de k beste kandidaat-sequenties bij. Bij elke tijdsstap wordt elke hypothese uitgebreid met een karakter, wat bij een uitvoerkarakterset van lengte V voor $k \cdot V$ extra sequenties zorgt. Deze worden toegevoegd aan de kandidaat-sequenties en opnieuw bewaren we enkel de k beste. De waarde k is hier de *beam size*. Wanneer $k = 1$ spreken we van *greedy inference*, de argmax die eerder besproken werd. Een sequentie wordt niet meer uitgebreid wanneer de decoder een speciaal *end of sequence* `<eos>`-token heeft voorspeld. *Beam search* stopt wanneer geen betere sequenties dan de huidige k kandidaat-sequenties gevonden worden.

6 RNN met attentionmechanisme

De encoder-decoderarchitectuur besproken in vorig hoofdstuk is een krachtig alternatief wanneer er geen beschikking is over alignering van de grafeem- en foneemparen. Op de publieke CMU-Dict dataset voorspelt de architectuur slechts 28,61% van de woorden fout, waar LSTM's met alignment 23,55% halen (Yao and Zweig, 2015). Uit analyse blijkt dat de meeste fouten gemaakt worden op lange woorden, omdat er meer info in de contextvector met beperkte constante lengte moet bewaard worden. Een toevoeging op de encoder-decoderarchitectuur, initieel voorgesteld door Bahdanau et al. (2014) en later uitgebreid door Luong et al. (2015), introduceert een **attentionmechanisme**. Het doel is om niet de volledige context te gebruiken bij het voorspellen van een foneem, maar te *leren* welke grafemen een rol spelen bij het voorspellen van een foneem. In plaats van de invoersequentie $\mathbf{x} = (x_1, x_2, \ldots, x_{T_x})$ voor te stellen in één contextvector met vaste lengte $\mathbf{c}$, worden alle hidden states van de encoder gecombineerd om voor elke tijdsstap een unieke contextvector $\mathbf{c}_t$ te genereren. We bespreken twee types attention: globaal en lokaal. De mechanismen zijn ongeveer gelijk, met het verschil dat globale attention toegang heeft tot alle T_x hidden states, waar lokale attention slechts een beperkt venster in rekening brengt.

6.1 Globale attention

In de encoderstap verandert er weinig ten opzichte van een model zonder attention, behalve dat de hidden states $\mathbf{h}_s$ van elke tijdsstap nu bijgehouden worden. De decoder werkt iets anders. Op elke tijdsstap t wordt een contextvector $\mathbf{c}_t$ berekend op basis van alle hidden states van de encoder. De berekening gebeurt via een gewogen som, waarbij de gewichten $a_{t,s}$ aanduiden hoeveel *aandacht* er besteed moet worden aan hidden state $\mathbf{h}_s$ op tijdstip t:

$$\mathbf{c}_t = \sum_{s=1}^{T_x} a_{t,s} \mathbf{h}_s.$$

Merk op dat dit conceptueel overeenkomt met een alignering van invoer- en uitvoersequentie. De alignering is niet eenduidig zoals de many-to-many alignment uit Sectie 4.1, maar kent een continu verloop. Grafemen kunnen bijvoorbeeld voor 70% met één foneem gealigneerd worden en 30% met een ander. We spreken daarom van een **soft-alignment**.

De contextvector wordt vervolgens in de decoder gebruikt in een extra **attention-laag**, om een finale uitvoer $\tilde{\mathbf{h}}_t$ te verkrijgen (die vervolgens een *softmax* ondergaat). Er zijn verschillende manieren waarop de contextvector $\mathbf{c}_t$ met de uitvoer van de decoder $\mathbf{h}_t$ gecombineerd kan worden. Bahdanau et al. (2014) gebruikt de contextvector geconcateneerd met de vorige hidden state in een extra RNN-laag. Luong et al. (2015) combineert de contextvector met de uitvoer van de decoder in een eenvoudige concatenatie:

$$\tilde{\mathbf{h}}_t = \begin{cases} \text{RNN}([\mathbf{c}_t; \mathbf{h}_{t-1}]) & \textit{Bahdanau} \\ \tanh(W_c[\mathbf{c}_t; \mathbf{h}_t]) & \textit{Luong.} \end{cases}$$

Tot slot wordt de berekening van de attention-gewichten $a_{t,s}$ besproken, welke uiteindelijk zullen bepalen hoe de contextvector eruit ziet. De attentiongewichten $a_{t,s}$ van stap t en hidden state $\mathbf{h}_s$ worden berekend via een *softmax* van attentionscores tussen de hidden state van de decoder

$\mathbf{h}_t$ en de hidden state van de encoder $\mathbf{h}_s$:

$$a_{t,s} = \frac{\exp\left(\text{Score}(\mathbf{h}_t, \mathbf{h}_s)\right)}{\sum_S \exp\left(\text{Score}(\mathbf{h}_t, \mathbf{h}_s)\right)}.$$

Voor de scores zijn er verschillende opties:

$$\text{Score}(\mathbf{h}_t, \mathbf{h}_s) = \begin{cases} \mathbf{h}_t^T \mathbf{h}_s & \textit{dot} \\ \mathbf{h}_t^T W_a \mathbf{h}_s & \textit{general} \\ \mathbf{v}_a^T \tanh(W_a[\mathbf{h}_t; \mathbf{h}_s]) & \textit{concat.} \end{cases}$$

Bahdanau-attention gebruikt enkel de concatenatie, waarbij de vorige staat $\mathbf{h}_{t-1}$ gebruikt wordt (omdat $\mathbf{h}_t$ zelf berekend wordt uit de attentionscores). De gewichten W_c, W_a en $\mathbf{v}_a$ worden samen met de encoder-decoder als een geheel getraind, waardoor de attentionlaag de soft-alignment op zichzelf leert.

Figuren 11 en 12 stellen de subtiele verschillen schematisch voor. Waar Bahdanau klassieke teacher forcing gebruikt, maakt Luong gebruik van een **input-feeding approach**, waarbij de uitvoerlogits $\tilde{\mathbf{h}}_t$ geconcateneerd worden aan de targetinput om als invoer in de decoder te gebruiken. Zo is het model zich volledig gewaar van de vorige alignment-keuzes.

6.2 Lokale attention

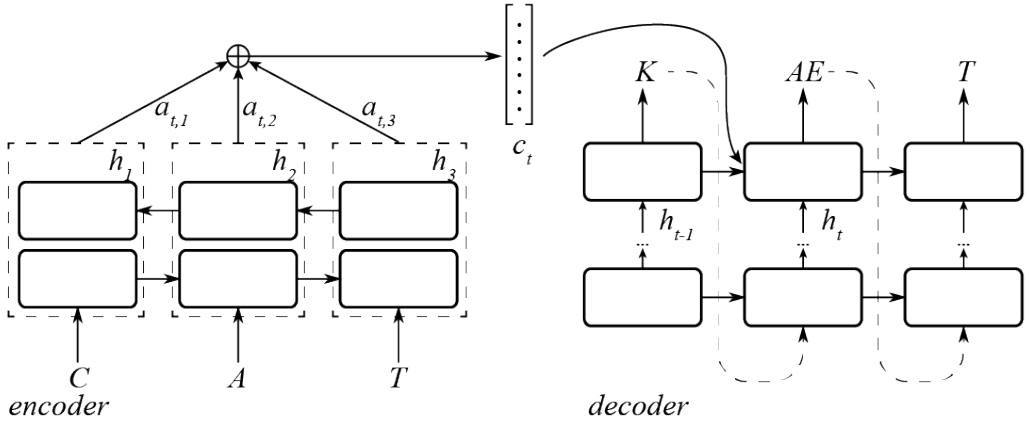
Globale attention heeft het nadeel dat de attentionlaag aandacht moet schenken aan de volledige invoersequentie, wat een dure operatie is voor langere sequenties zoals bij MT of G2P op zinsniveau. Dit nadeel kan omzeild worden door slechts een deelsequentie van de invoer te bekijken met een principe van lokale attention (Luong et al., 2015). Op elke tijdsstap t bekijkt lokale attention enkel de hidden states in een window $[p_t - \frac{w}{2}; p_t + \frac{w}{2}]$, waarbij de breedte w van het window empirisch gekozen wordt en de variabele p_t de alignment-positie is. Voor de keuze van p_t hebben we twee opties:

Monotonic alignment – We stellen $p_t = t$, ervan uitgaande dat de invoer- en uitvoersequenties ongeveer dezelfde lengte hebben. Deze alignment kan zeer goed werken, maar is sterk afhankelijk van de use case.

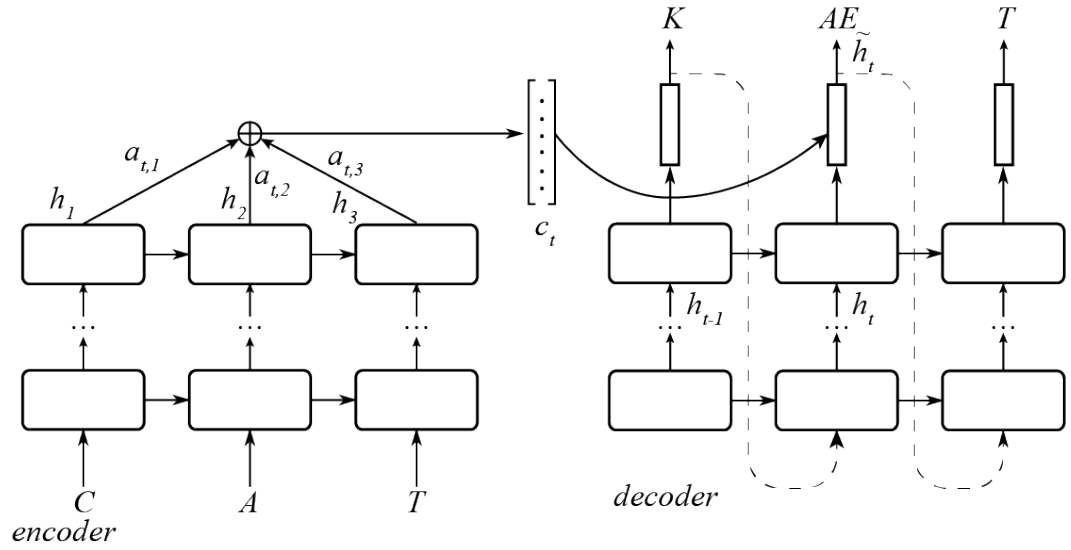
Predictive alignment – We laten het model de alignering zelf leren, op basis van:

$$p_t = s \cdot \sigma(\mathbf{v}_p^T \tanh(W_p \mathbf{h_t})),$$

waarbij s, $\mathbf{v}_p$ en W_p trainbaar zijn.



Figuur 11: Een encoder-decoderarchitectuur met Bahdanau attention. De originele paper van Bahdanau et al. (2014) gebruikt een BLSTM-stack met één forward en één backward pass, die geconcateneerd worden tot $\mathbf{h}_t$.

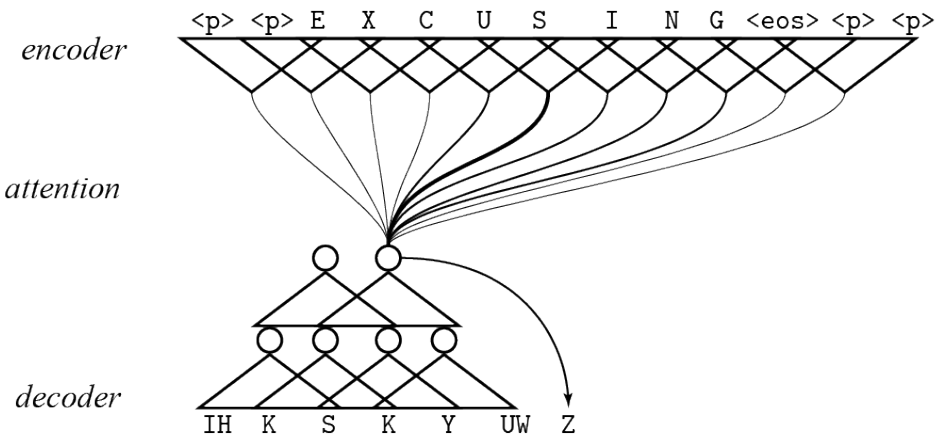


Figuur 12: Een encoder-decoderarchitectuur met Luong attention. De $a_{t,s}$-waarden worden hier berekend op basis van $\mathbf{h}_t$ in tegenstelling tot $\mathbf{h}_{t-1}$ bij Bahdanau.

7 Convolutionele attention

RNN-gebaseerde modellen met attention hebben bewezen dat ze zeer krachtig zijn voor machine translation en andere NLP-problemen. Ze hebben echter een nadeel dat de trainingstijd drastisch beïnvloedt: ze lenen zich niet eenvoudig tot parallellisatie. Per definitie heeft elke RNN-cel nood aan de hidden state van de vorige. De berekeningen van alle volgende cellen moeten wachten zolang de hidden state er niet is. Hierdoor is de snelheid van training afhankelijk van de lengte van de invoer- en uitvoersequenties. LSTM's zijn bovendien computationeel zeer intensief. Ze hebben veel trainbare parameters en vragen bijgevolg veel geheugen, vooral tijdens backpropagation through time. Door de steeds efficiënter wordende parallelle rekenkracht van GPU's en TPU's, willen we daar ook gebruik van maken door zo veel mogelijk taken parallel uit te voeren.

Het **convolutionele sequence-to-sequence model** van Gehring et al. (2017) probeert de encodering zo veel mogelijk te parallelliseren door de RNN-cellen te vervangen door een convolutioneel neuraal netwerk (CNN). Convoluties combineren een vast aantal tokens tot een vectorrepresentatie via een zeer snelle bewerking die zich goed tot parallellisatie leent. Ze kunnen hierdoor de hardware en optimalisaties van de hedendaagse GPU's beter exploiteren.



Figuur 13: Architectuur van ConvS2S (Gehring et al., 2017), toegepast op G2P. Op elke tijdsstap wordt de hypothese bepaald op basis van de convolutie van de vorige voorspellingen en de attentionvector.

7.1 Convolutionele blok

Zowel de encoder en decoder zijn opgebouwd uit op elkaar gestapelde convolutionele blokken. De blokken worden vaak (al dan niet gecombineerd) voorgesteld als een driehoek, zoals in Figuur 13. Elke blok voert eerst een eendimensionale convolutie uit, gevolgd door een niet-lineaire activatie en een residual connection, waarna de uitvoer naar de volgende blok wordt gestuurd.

Een **eendimensionale convolutie** is een manier om een vast aantal inputelementen te combineren tot een kleinere representatie. Dit gebeurt door een matrix van gewichten W, de kernel, over de inputelementen te schuiven en telkens het scalair product te nemen van de kernel en de elementen die onder de kernel liggen. De invoer X voor een character embedding van dimensie d is een matrix met dimensies $T_x \times d$. Indien we convolueren met een filterbreedte k, verkrijgen

we bij elk convolutioneel blok i een scalair getal dat bepaald wordt door onderstaande formule:

$$\text{conv}(X, i) = \sum_d \mathbf{w}_d \cdot \mathbf{x}_{(i-\frac{k}{2}):(i+\frac{k}{2}),d}$$

Een convolutioneel blok volgens Gehring et al. (2017) voert niet één convolutie uit, maar $2d$ verschillende convoluties, telkens met een andere gewichtenmatrix W.
Na de convolutie wordt de verkregen vector met lengte $2d$ in twee gesplitst en door een **Gated Linear Unit** (GLU) gestuurd. Een GLU is een niet-lineair mechanisme dat de dimensie van een vector halveert via:

$$\text{GLU}(\begin{bmatrix} A & B \end{bmatrix}) = A \circ \sigma(B).$$

De output van de GLU wordt niet rechtstreeks doorgestuurd naar de volgende convolutionele blok, maar ze wordt eerst gesommeerd met de initiële input $\mathbf{h}_i^{[l-1]}$ van de blok:

$$\mathbf{h}_i^{[l]} = \mathbf{h}_i^{[l-1]} + \text{GLU}(W^{[l]}\mathbf{h}^{[l-1]}_{(i-\frac{k}{2}):(i+\frac{k}{2})} + \mathbf{b}_w^{[l]}).$$

Een dergelijke verbinding heet een **residual connection** en is een manier om neurale netwerken eenvoudiger te trainen.

7.2 Positionele embedding

Net zoals RNN's maken CNN's ook gebruik van input embedding om beter om te kunnen gaan met grote alfabetten en symbolen met verwante betekenis. In tegenstelling tot RNN's kunnen convolutionele operaties echter niet rechtstreeks gebruik maken van de positie van het symbool in de inputsequentie. Zonder positionele embedding ziet een convolutieoperatie bijvoorbeeld geen verschil tussen "eat" en "ate". Om dit probleem aan te pakken, wordt een embeddingmatrix W_{pos} opgesteld voor de absolute positie van het symbool, op analoge manier als character embedding (Sectie 3.1). Deze embedding wordt dan opgeteld bij de character embedding:

$$E_{ch} = X_{onehot} \cdot W_{ch}$$

$$E_{pos} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 2 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & T_x \end{bmatrix} \cdot W_{pos}$$

$$X = E_{ch} + E_{pos}$$

De positionele embedding wordt toegepast op zowel de input- als de targetsequentie.

7.3 Attentionmechanisme

De attention bij een CNN-model gebeurt in elk decoderblok. De structuur van de decoder is analoog aan de encoder, alleen heeft die enkel toegang tot de reeds voorspelde symbolen. Op het einde van elk decoderblok wordt via een attentionmechanisme de info uit de inputsequentie

toegevoegd. Eerst wordt een *decoder summary* $\mathbf{d}_i$ berekend op basis van de huidige decoder state h_i en het laatst voorspelde element $\mathbf{y}_{i-1}$:

$$\mathbf{d}_i = W_d \mathbf{h}_i + \mathbf{b}_d + \mathbf{y}_{i-1}$$

met W_d en $\mathbf{b}_d$ trainbare parameters. De attentionscore a_{ij} voor decoderblok i en invoerelement j wordt berekend als een *softmax* over het scalair product tussen de decoder summary $\mathbf{d}_i$ en de uitvoer van het encoderblok $\mathbf{z}_j$. Die attentionscore wordt dan samen met de originele input $\mathbf{x}_j$ gebruikt om de contextvector $\mathbf{c}_i$ te berekenen:

$$a_{ij} = \frac{\exp\left(\mathbf{d}_i \cdot \mathbf{z}_j\right)}{\sum_t \exp\left(\mathbf{d}_i \cdot \mathbf{z}_t\right)}$$

$$\mathbf{c}_i = \sum_{j=1}^{T_x} a_{ij}(\mathbf{z}_j + \mathbf{x}_j)$$

De contextvector wordt in elke convolutieblok als invoer meegegeven.
Door de convolutieblokken te stapelen kunnen zowel lokale als long-range relaties gemodelleerd worden. Voor een kernelbreedte k kunnen we een sequentie van n woorden tot een contextvector modelleren met $O(n/k)$ convolutionele operaties, in tegenstelling tot RNN's, die $O(n)$ operaties nodig hebben. Uit experimenten blijkt ook dat convolutionele encoder-decoders zoals ConvS2S (Gehring et al., 2017) en ByteNet (Kalchbrenner et al., 2016) een betere nauwkeurigheid halen op de MT-taak dan gelijkaardige RNN-modellen.

8 Transformer attention

Het sequentiële karakter van RNN's maakte het moeilijker om ten volle gebruik te maken van moderne GPU's en TPU's, die uitblinken in parallelle berekeningen. Convolutionele neurale netwerken zijn veel minder sequentieel dan RNN's, maar ze zijn gelimiteerd in de afstand van hun dependencies. Het aantal lagen (stapels convolutionele blokken) dat nodig is om informatie uit ver uiteenliggende delen van de invoer te combineren stijgt recht evenredig met de afstand tussen die delen. Hoe verder uit elkaar belangrijke delen van de invoer liggen, hoe meer lagen het CNN nodig heeft om deze relatie te modelleren.
Een relatie tussen elementen uit de sequenties noemen we **dependencies**. Dependencies kunnen opgedeeld worden in drie groepen (Kurita, 2017): relaties tussen

1. De invoer- en uitvoersymbolen
2. De invoersymbolen onderling
3. De uitvoersymbolen onderling

RNN en CNN attentionmechanismen lossen het eerste probleem (de grafeem-foneemalignering) op door de decoder toegang te geven tot de volledige invoersequentie. De andere twee, *self-attention* of *intra-attention* genoemd, worden er genegeerd. Toch geven ze belangrijk inzicht om een NLP-taak beter op te lossen.

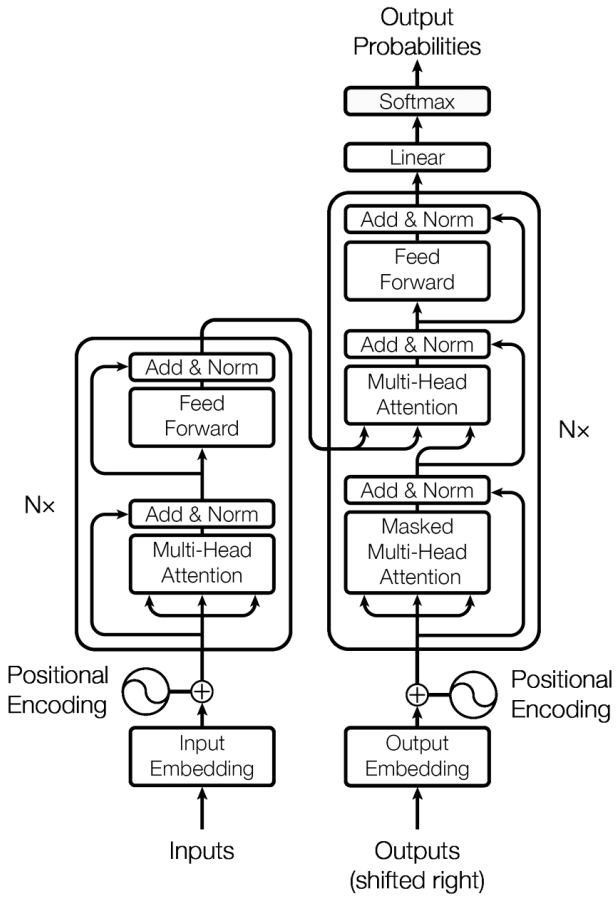
Vaswani et al. (2017) proberen zowel parallellisatie als long-range dependencies aan te pakken in hun paper "Attention is All You Need", door een nieuwe architectuur voor te stellen: de Transformer. Het Transformermodel bereikt parallellisatie door het sequentiële aspect volledig weg te laten en de dependencies louter met self-attention te modelleren. Er wordt nog steeds met een encoder-decoderarchitectuur gewerkt, waarin de invoersequentie en de targetsequentie (*teacher forcing*) worden ingegeven en een probabiliteitsdistributie per tijdsstap wordt teruggegeven. Figuur 14 geeft een vereenvoudigde voorstelling van de Transformer weer, zoals voorgesteld door Vaswani et al. (2017).

8.1 Encoder

De encoder is in zijn geheel een stack van N kleinere encoderblokken. De onderste encoder krijgt de character embedded vector van de invoersequentie binnen, de andere een steeds meer geëncodeerde versie ervan. Het idee hier is dat dicht bij elkaar staande invoerelementen in de lagere blokken met elkaar interageren om de lokale dependencies te vinden, terwijl long-range dependencies in de hogere blokken worden opgevangen. De encoderblokken zelf bestaan uit twee lagen. De eerste is een **multi-headed self-attentionlaag**, die voor elk symbool uit de sequentie een self-attentionvector berekent en ze doorstuurt naar de tweede laag, een klassieke **feedforward laag**.

De berekening van de self-attentionvector $\mathbf{z}_i$ gebeurt in drie stappen (Alammar, 2018):

In een eerste stap wordt voor elk symbool i een query-vector $\mathbf{q}_i$, een key-vector $\mathbf{k}_i$ en een value-vector $\mathbf{v}_i$ berekend door de inputvector van het encoderblok te vermenigvuldigen met een gewichtenmatrix. De gewichtenmatrix wordt net zoals bij character embedding mee getraind



Figuur 14: De architectuur van de Transformer (Vaswani et al., 2017). De encoder- (links) en decoder- (rechts) cellen worden N maal herhaald.

met de rest van het model. De betekenis van die drie vectoren wordt duidelijk wanneer het geheel behandeld is.
De tweede stap berekent voor elk koppel invoersymbolen (i, j) een self-attentionscore. Deze score bepaalt hoe gerelateerd een symbool is aan een ander symbool in de sequentie. De score wordt berekend uit het scalair product van $\mathbf{q}_i$ en $\mathbf{k}_j$. Vervolgens wordt ze geschaald op basis van de dimensie van de key-vector en genormaliseerd met een *softmax*-functie:

$$\text{Score}(\mathbf{q}_i, \mathbf{k}_j) = \text{softmax}(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}).$$

De schaling met factor $\sqrt{d_k}$, de dimensie van $\mathbf{k}_j$, zorgt voor een stabielere gradient descent.
De attentionscore wordt in een laatste stap gebruikt om een gewogen gemiddelde van alle value-vectoren op te stellen. Deze vector $\mathbf{z}_i$ zal in het feedforward netwerk gestuurd worden om de uiteindelijke attentionvector te bekomen.

$$\mathbf{z}_i = \sum_j \text{Score}(\mathbf{q}_i, \mathbf{k}_j)\mathbf{v}_j.$$

In de praktijk zullen we deze self-attentionvector niet één per één voor alle symbolen uit de sequentie berekenen, want dan zou er geen voordeel zijn ten opzichte van een RNN. De kracht van de Transformer is dat hij alle berekeningen tegelijk kan uitvoeren door eenvoudige matrix-multiplicatie toe te passen. De invoersequentie wordt voorgesteld in één matrix X. Elke rij in de matrix komt overeen met de embedding van een element:

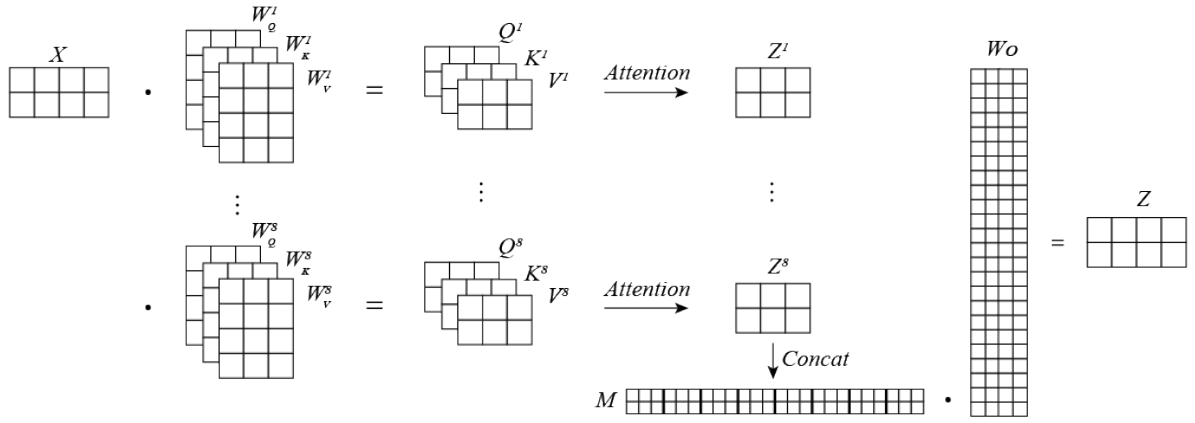
$$\begin{aligned}
Q &= W_Q X && X \in \mathbb{R}^{T_x \times d_{emb}}, W_Q \in \mathbb{R}^{d_{emb} \times d_k} \\
K &= W_K X && W_K \in \mathbb{R}^{d_{emb} \times d_k} \\
V &= W_V X && W_V \in \mathbb{R}^{d_{emb} \times d_v} \\
Z &= \text{Attention}(Q, K, V) = \text{softmax}(\frac{Q \cdot K^T}{\sqrt{d_k}})V. && (10)
\end{aligned}$$

Om de self-attentionlaag nog krachtiger te maken, voegt de paper een extra **multi-headed** mechanisme toe. In plaats van één attentionmatrix Z te berekenen, worden er $h = 8$ verschillende versies van Z berekend, elk met andere gewichtenmatrices. Op die manier leert de Transformer verschillende representaties voor de invoersequentie, waarop die telkens een self-attention toepast. Deze zogenaamde *attention heads* Z_i worden horizontaal geconcateneerd om één brede multi-headed attentionvector M te bekomen. Vervolgens wordt M vermenigvuldigd met een trainbare matrix W_O om de dimensies van Z_i te bekomen, zodat de hogere encoderblokken dezelfde dimensies kunnen hanteren. In onderstaande vergelijkingen en in Figuur 15 worden de dimensies van de verschillende matrices duidelijk:

$$\begin{aligned}
Z_i &= \text{Attention}(W_Q^i X, W_K^i X, W_V^i X) && Z_i \in \mathbb{R}^{T_x \times d_v} \\
M &= \text{Concat}(Z_1, \ldots, Z_h) && M \in \mathbb{R}^{d_{emb} \times h d_k} \\
Z &= MW_O && W_O \in \mathbb{R}^{h d_v \times d_{emb}}, Z \in \mathbb{R}^{d_{emb} \times d_k}.
\end{aligned}$$

De output van de self-attentionlaag en de feedforwardlaag wordt niet rechtstreeks doorgestuurd naar de volgende encoderblok, maar ze wordt via een *residual connection* gesommeerd met de

initiële input van die laag. De residual connection wordt gevolgd door een *layer normalisation*, een normalisatiestrategie.



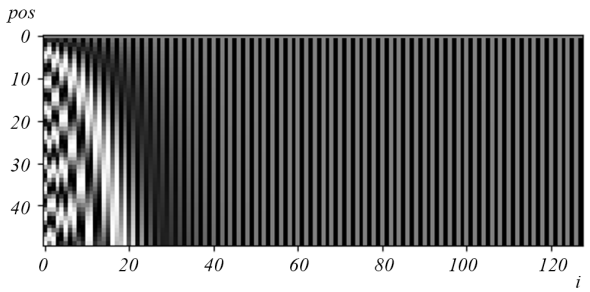
Figuur 15: Visuele voorstelling van de constructie van een multi-headed self-attentionmatrix Z met 8 attention-heads. Merk op dat de lengte van de sequentie $d_{seq} = 2$, de embedding dimensie $d_{emb} = 4$ en de key- en valuedimensies $d_k = d_v = 3$.

8.2 Positional encoding

Net zoals bij CNN's kan de self-attention niet rechtstreeks gebruik maken van de positie van het symbool in de inputsequentie. De Transformer gebruikt hierbij geen trainbare embeddingmatrix (Sectie 7.2), maar ze gebruikt een vaste encoderingsfunctie. De initiële embeddingvector wordt gesommeerd met een *positional encoding* PE. De encoding wordt gedefinieerd door:

$$\text{PE}(pos) = \begin{cases} \sin(pos/10000^{2i/d_{emb}}) & \text{voor even } i \leq d_{emb} \\ \cos(pos/10000^{2i/d_{emb}}) & \text{voor oneven } i \leq d_{emb} \end{cases}$$

waarbij *pos* de positie is van het symbool in de sequentie en i de dimensie (de index) in de embeddingruimte. De vectoren kunnen op voorhand berekend worden indien de lengte van de langste sequentie gekend is. Figuur 16 geeft een visualisatie van de positionele encoding in functie van *pos* en i. De bedoeling is dat de Transformer het patroon zelf leert en het gebruikt om de positie van elk symbool of de afstand tussen twee symbolen te bepalen.



Figuur 16: Positional encodings voor een $d_{emb} = 128$ en alle *pos* tussen 0 en 50. Hoe groter de (eventueel negatieve) waarde, hoe lichter de grijswaarde.

8.3 Decoder

De structuur van de decoder is zeer gelijkaardig aan die van de encoder, maar ze heeft een extra laag tussen de self-attention en de feedforwardlaag. Bovendien produceert ze slechts één symbool per iteratie.

Eerst verwerkt ze de uitvoer van een lager decoderblok in een **masked self-attentionlaag**. Hierbij kan de decoder enkel eerdere symbolen uit de targetsequentie bekijken. Dit wordt verwezenlijkt door toekomstige posities weg te maskeren vóór de *softmax*-stap in (10).

Daarna wordt de self-attentionvector Z gecombineerd met de uitvoer van de encoder in een tweede attentionlaag, de **encoder-decoder attentionlaag**. Deze gebruikt een verzameling key en value-matrices K_{enc} en V_{enc} verkregen uit de attentionvector van het laatste encoderblok, samen met de querymatrix uit het decoderblok.

8.4 Lineaire en softmax-laag

De output van de laatste decoderblok is een vector met vaste lengte. De taak van de lineaire laag is om deze vector om te zetten naar een logit vector over een alfabet met eventueel andere dimensies. De lineaire laag is een klassiek fully-connected netwerk. Vervolgens wordt de logit vector genormaliseerd door een *softmax*-laag en kan greedy of beam search toegepast worden om een uitvoerhypothese te selecteren.

9 Experimenten

In de vorige hoofdstukken hebben we verschillende categorieën van encoder-decoderarchitecturen besproken. Publicaties bewijzen dat sommige architecturen beter presteren dan andere op bepaalde Engelstalige datasets, maar elke taal is immers verschillend en de performantie van de verschillende modellen op andere datasets is ongekend. Bovendien kan de keuze van het foneemalfabet de complexiteit van het G2P-probleem sterk verhogen/verlagen.

In deze sectie wordt de opstelling besproken in het kader van de verschillende experimenten. Eerst worden de gebruikte datasets en foneemalfabetten nader bekeken. Vervolgens bespreken we de gebruikte metrieken voor evaluatie en vergelijking van de modellen onderling. Verder wordt naar de gebruikte open source toolkits verwezen om tot slot de trainingsdetails te bekijken.
In het volgende hoofdstuk analyseren we de resultaten.

9.1 Datasets

Voor de experimenten worden vier datasets gebruikt. Drie datasets voor woordniveau en een vierde dataset voor training of zinsniveau. De eerste is de CMUDict dataset[2]. Deze publieke dataset bevat 126 191 Amerikaans Engelse woorden met hun bijhorende transcriptie. Ze wordt dikwijls gekozen om resultaten te vergelijken met de huidige state-of-the-art modellen. Net zoals Rao et al. (2015); Toshniwal and Livescu (2016); Yolchuyeva et al. (2019); Yao and Zweig (2015); Galescu and Allen (2002), wordt een trainingsset van 113 438 en een testset van 12 000 woorden gebruikt. Deze testset bevat dezelfde samples als de literatuur, zodat de cijfers rechtstreeks vergeleken kunnen worden. Uit de trainingsset worden 5 660 samples weggehouden als validatie. De transcripties maken gebruik van het eerder besproken ARPABET, maar de primaire en secundaire klemtoonsuffixen worden weggelaten. Zo wordt er een grafeemalfabet van 27 karakters (alle letters uit het Latijns alfabet en een apostrof) en een foneemalfabet van 39 fonemen verkregen.

De andere twee datasets voor woordniveau bevatten in-house Turkse en Kroatische transcripties. De Turkse dataset bevat 76 082 samples en wordt aangevuld met 8 589 geografische eigennamen, 5 080 voornamen en 17 179 familienamen. De Kroatische dataset bevat 109 006 samples. Informatie over homografen is weergegeven in Tabel 2.

Tabel 2: Aantal homografen (woorden met > 1 uitspraak) per dataset.

Dataset	**Woorden**	**Homografen**
CMUDict	126 191	6 550 (5,19%)
Turks	106 457	327 (0,31%)
Kroatisch	107 150	3 460 (3,23%)

De laatste dataset bevat transcripties op zinsniveau. Uit een reeks Engelstalige corpora met verschillende thema's waaronder nieuws, sport en wetenschap worden 50 000 zinnen willekeurig gekozen. Deze worden aangevuld met een verzameling van specifiek geselecteerde zinnen die een homograaf bevatten, waarvan de uitspraak in de betreffende context gekend is. Na verwijdering

[2]Met dank aan Stanley Chen voor het verstrekken van de data.

van duplicaten levert dit een corpus op van 67 998 zinnen op (1 524 430 woorden, waarvan 65 528 verschillende orthografieën). Bij gebrek aan manueel gelabelde data worden de corpora omgezet naar fonemen via het huidige G2P-systeem. Dit leidt uiteraard tot een suboptimaal geheel: door de fouten in het huidige systeem kunnen verkeerde verbanden geleerd worden. De homografen uit de daarvoor geselecteerde zinnen worden gecontroleerd en eventueel overschreven door hun correcte uitspraak, zodat die zeker correct getraind worden.

Om character embedding en one-hot encoding te kunnen toepassen, moeten op voorhand een eindig grafeem- en foneemalfabet gedefinieerd worden.

De in-house datasets maken gebruik van het L&H+ foneemalfabet, een zeer rijk foneemalfabet vergelijkbaar aan IPA, dat onder meer prosodische informatie bevat. De aanduiding voor deze prosodische, zogenaamde *suprasegmentale* symbolen is vrij analoog aan IPA. Ze bevatten onder meer:

Lettergreepgrenzen – aangeduid door een "`.`" en dienen als extra metadata voor de synthesizer. Lettergreepgrenzen hebben invloed op de klemtoon, de snelheid waarmee fonemen worden uitgesproken en de micropauzes tussen fonemen.

Primaire klemtoon – aangeduid door "`'`" aan het begin van een lettergreep. L&H+ definieert ook secundaire klemtoon "`'2`" en zelfs tertiaire klemtoon "`'3`", maar datasets die deze informatie bevatten zijn zeldzaam.

Duratie van de klinker – aangeduid door "`:`" indien ze langer dan normaal moet worden uitgesproken.

Tooncontouren aangeduid door o.a. "`15`" en "`51`". Tooncontouren komen voor wanneer de toonhoogte verandert tijdens het uitspreken van een klinker.

Deze bovenstaande tekens hebben allemaal te maken met prosodie: het ritme, de klemtoon, toonhoogte en de intonatie van de stem bij het uitspreken van lettergrepen en grotere spraakeenheden. In TTS is prosodie een belangrijk aspect om de stem niet robotisch te laten overkomen. Tabel 3 toont enkele voorbeelden uit de Kroatische dataset, waar de aanwezigheid van suprasegmentalen te zien is.

Tabel 3: Enkele voorbeelden uit de Kroatische dataset.

Orthografie	**Fonetische transcriptie**
šulkolega	`'Su:51l.ko.le:.ga`
epileptičarka	`e.pi.'le15.pti.t&Sa:r.ka`
pjegavac	`'pje51.ga.Vat&s`
uvakufiti	`'u15.Va.ku.fi.ti`
omatati	`o.'ma:15.ta.ti`

Elk L&H+-foneem bestaat uit een of meerdere ASCII-karakters, zoals de `u:51` (een lange u met dalende toonhoogte) en de `t&S` (een CH-klank). Een eerste stap moet de fonetische transcriptie inlezen en splitsen in enkelvoudige fonemen, zodat het neuraal netwerk weet welke karakters een foneem vormen. Dit gebeurt via een a priori opgesteld foneemalfabet en enkele reguliere expressies. Fonemen die niet voorkomen in een taal zullen niet worden opgenomen in het betreffend alfabet. Het proces dat de transcriptie splitst in foneemsymbolen wordt de *tokenization* genoemd.

Tabel 4 illustreert een tokenization van de voorbeelden uit Tabel 3. De lettergreepgrenzen en klemtoon zullen als aparte fonemen beschouwd worden, maar duratie en tooncontouren worden mee met de voorafgaande klinker voorspeld. Zo krijgen we bijvoorbeeld aparte fonemen voor `e`, `e15`, `e51`, `e:`, `e:15` en `e:51`. Op basis van deze conventie heeft het Turks 54 verschillende fonemen en voor het Kroatisch 81, waaraan speciale `<sos>` en `<eos>`-tokens worden toegevoegd om het begin en einde van een woord te markeren.

Tabel 4: Tokenization van de voorbeelden uit Tabel 3.

Tokenized transcriptie
’ S u:51 l . k o . l e: . g a
e . p i . ’ l e15 . p t i . t&S a: r . k a
’ p j e51 . g a . V a t&s
’ u15 . V a . k u . f i . t i
o . ’ m a:15 . t a . t i

Om een grafeemalfabet op te stellen, wordt er tijdens het inlezen van de onbewerkte data een *unordered set* bijgehouden met alle voorkomende grafemen. Hoofdletters worden er niet genormaliseerd, omdat ze extra duiding kunnen geven aan de uitspraak van eigennamen. Bovendien kan de character-embeddinglaag zelf leren dat letters en hun hoofdlettervariant gelijkaardige betekenis hebben en zullen hun embeddingvectoren relatief dicht bijeen liggen. Wel worden eventuele spaties vervangen door underscores om problemen in de embedding te vermijden. De grafeemalfabetten verschillen ten opzichte van de CMUDict, met 29 grafemen voor het Kroatisch en 89 voor het Turks. Het Turkse schrift gebruikt een uitgebreid alfabet met extra letters zoals ç, ğ, ı, ö, ş en ü.
Vervolgens worden duplicaten (met exact dezelfde grafeem- en foneemsequentie) uit de datasets verwijderd, worden de alfabetisch gerangschikte samples random dooreen geschud en wordt er een train-test-val split gemaakt met ratio 80%-10%-10%. Een vereenvoudigd script voor datavoorbereiding, geschreven in Python is weergegeven in Bijlage A.

9.2 Evaluatie

De kwaliteit van een G2P-model wordt steeds uitgedrukt in Phoneme Error Rate (PER) en Word Error Rate (WER) van de voorspelde transcriptie tegenover de werkelijke transcriptie.

Phoneme Error Rate

Intuïtief zou men als Phoneme Error Rate de voorspelde fonemen één voor één vergelijken met de verwachte fonemen, maar dit heeft zijn beperkingen. Wanneer een G2P model bijvoorbeeld als enige fout een extra teken tussenvoegt waar er geen zou moeten staan, zijn alle daarop volgende tekens bijgevolg ook fout (omdat ze één plaats verder staan). Er wordt dus gekozen voor een metriek die ook tussenvoegingen en verwijderingen van karakters ondersteunt, zonder die fouten verder in de sequentie te accumuleren.
De Levenshteinafstand (of *edit distance*, Levenshtein (1966)) is het minimale aantal bewerkingen (karakters invoegen, verwijderen of veranderen) om de voorspelde string s te transformeren tot de verwachte string t. Ze kan gezien worden als een uitbreiding op het zoeken naar

de langste gemeenschappelijke deelsequentie (LCS), die enkel invoegen en verwijderen toelaat. Beide zijn NP-complete problemen die met dynamisch programmeren kunnen opgelost worden. Codevoorbeeld 1 toont de implementatie in Python met behulp van Numpy. Ze berekent de Levenshteinafstand tussen alle mogelijke prefixen van beide woorden (inclusief een lege string), waardoor een matrix met dimensies $l_s + 1 \times l_t + 1$ van afstanden wordt opgesteld, met l_s en l_t de lengtes van string s en t. De Levenshteinafstand tussen beide woorden bevindt zich dan in de rechteronderhoek van die matrix. Het codevoorbeeld gebruikt een extra geheugenoptimalisatie waarbij we enkel de huidige en voorgaande rij uit de matrix bijhouden. Hierdoor heeft ze een geheugenefficiëntie van $O(l_t)$ en een tijdsefficiëntie van $O(l_s l_t)$. Indien we ook willen weten welke fouten er gemaakt worden (invoegen, verwijderen of veranderen), moet de volledige matrix samen met een *backtrace*-matrix bijgehouden worden. De code hiervoor is weergegeven in Bijlage B.
Als metriek voor de PER gebruiken we de Levenshteinafstand in verhouding met het totaal aantal verwachte tekens.

Codevoorbeeld 1: Levenshteinafstand.

```
def levenshtein(s, t):
    if s == t:
        return 0
    elif len(s) == 0:
        return len(t)  # T insertions
    elif len(t) == 0:
        return len(s)  # S deletions

    v0 = np.arange(len(t) + 1)  # compares t to an empty string
    v1 = np.empty(len(t) + 1)

    for i in range(len(s)):
        v1[0] = i + 1  # compares s to an empty string
        for j in range(len(t)):
            v1[j + 1] = min(v1[j] + 1,       # deletion
                            v0[j + 1] + 1,  # insertion
                            v0[j] + (0 if s[i] == t[j] else 1))
                            # substitution or correct
        v0 = np.copy(v1)

    return v1[-1]
```

Word Error Rate

Wanneer één foneem in een transcriptie fout is, wordt dit woord fout uitgesproken en is er een grote kans dat het door de gebruiker ook fout geïnterpreteerd zal worden. De WER geeft hier inzicht in en wordt berekend als de verhouding verkeerd voorspelde woorden op het totaal aantal woorden. Ze vertelt iets meer over de distributie van de PER, die bijvoorbeeld hoog is omdat sommige woorden veel fouten hebben, of omdat alle woorden één fout hebben.

Wanneer een getraind G2P-model een woordenboek moet converteren, is er geen contextinfor-

matie om homografen af te handelen. Daarom tolereren we bij de berekening van de WER dat het model een andere geldige uitspraak voorspelt dan de verwachte. Bij de berekening van de PER berekenen we de Levenshteinafstand tussen de voorspelling en alle mogelijke transcripties, om vervolgens de kleinste te bewaren. Het algoritme dat rekening houdt met homografen staat weergegeven in Bijlage C. Wanneer we een model evalueren op zinsniveau, zullen we alternatieve transcripties op homografen uiteraard niet tolereren.

9.3 Modellen

De verschillende topologieën waarmee geëxperimenteerd wordt zijn de RNN-gebaseerde encoder-decoderarchitectuur met en zonder attention, een CNN-architectuur volgens Gehring et al. (2017) en een Transformer volgens Vaswani et al. (2017).
Voor de **RNN-architectuur** (in deze scriptie verder afgekort als model A) wordt gebruik gemaakt van de "Tensorflow NMT-toolkit" (Luong et al., 2017). Deze toolkit is specifiek bedoeld om verschillende attentionmechanismen toe te passen op machine translation. De toolkit heeft zowel Luong- als Bahdanau-attention ter beschikking en hyperparameters zoals het aantal lagen, de embeddingdimensie en de learning rate kunnen zeer eenvoudig aangepast worden. Zoals eerder vermeld, kunnen we G2P als een speciaal MT-probleem beschouwen, door de letters als woorden te zien en de fonemen als vertaalde woorden. Er zijn enkele metrieken aanwezig voor MT zoals BLEU en ROUGE, maar de Word Error Rate en Phoneme Error Rate moeten zelf geïmplementeerd worden.
Voor de attentiongebaseerde **CNN-architectuur** (verder afgekort als model B) is de code van Gehring et al. (2017) publiek beschikbaar onder de naam "Fairseq"[3], maar ze is geschreven in het minder toegankelijke Lua. Ter vervanging wordt een in-house aanpassing op de NMT-toolkit gebruikt, die een Tensorflow-implementatie van Fairseq[4] integreert. Ze bevat dezelfde nuttige eigenschappen als de NMT-toolkit, met extra keuzes zoals het aantal convolutielagen en de dimensies van de convolutiefilters.
Voor het **Transformermodel** (afgekort als model C) wordt gebruik gemaakt van een compleet andere codebase: de "g2p-seq2seq"[5] ontwikkeld aan de Carnegie Mellon University (dezelfde onderzoeksgroep die CMUDict ter beschikking stelt). Dit project is een submodule van de Tensor2Tensor toolkit (Vaswani et al., 2018), specifiek ontwikkeld voor G2P. Ze laat toe om het aantal encoderblokken, de embeddingdimensie en het aantal self-attention heads in te stellen. Enkele resultaten op CMUDict worden vermeld in de repository, maar er zijn voor zover ons bekend geen publicaties of grondige experimenten met deze nieuw voorgestelde architectuur gemaakt.
De drie toolkits maken gebruik van de populaire deep learning library Tensorflow en bieden integratie met TensorBoard, waardoor alle geproduceerde metrieken tijdens het trainingsproces gevolgd kunnen worden.

9.4 Training

Alle modellen worden getraind op één NVIDIA® Tesla® K80 GPU tot geen verbetering in de nauwkeurigheid op de validatieset meer waargenomen wordt. Training duurt ongeveer 4 uur per

[3] `https://github.com/facebookresearch/fairseq`
[4] `https://github.com/tobyyouup/conv_seq2seq`
[5] `https://github.com/cmusphinx/g2p-seq2seq`

model, maar is sterk afhankelijk van de learning rate, de grootte van de dataset en het aantal trainbare parameters in het model.

Alle architecturen maken gebruik van teacher-forcing tijdens de trainingsfase. Tussen elke laag wordt ook een dropout-laag voorzien. Dropout zorgt ervoor dat neuronen tijdens één trainingsiteratie genegeerd worden, alsof ze weggevallen zijn. Dropout voorkomt overfitting, door de sterke afhankelijkheden van twee neuronen onderling te verminderen. Door die neuronen uit te schakelen, wordt het netwerk geforceerd om nieuwe verbanden te leggen en meer te generaliseren. Welke neuronen wegvallen is stochastisch en wordt gedefinieerd door de *keep probability* p of de *dropout rate* $1 - p$. We gebruiken een dropout rate van 0,20 voor alle modellen.

Bij alle drie de modellen wordt er gebruik gemaakt van bucketing en minibatching met $k = 5$ en $n = 128$ bij model A en B en $k = 6$, $n = 4096$ bij model C.

Als optimizer wordt Adam gebruikt. Een optimizer zorgt voor een stabielere en snellere convergentie door de learning rate voor individuele gewichten aan te passen. Gewichten die niet frequent geüpdatet worden, krijgen dynamisch een hogere learning rate toegewezen. Adam implementeert ook een mechanisme van momentum. Momentum kan vergeleken worden met een bal die van een helling rolt: een gewicht dat recent veel in een richting is aangepast, zal de volgende iteratie opnieuw in dezelfde richting worden aangepast om zo lokale minima te ontkomen. Adam vereist een ϵ-waarde, welke 10^{-4} gekozen wordt voor model A , 10^{-3} voor model B en 10^{-6} voor model C. De loss-functie die bepaalt hoe de gewichten aangepast worden, is in de drie modellen de cross-entropy loss L_{CE} uit (7).

9.5 Inference

Voor inference op de testset wordt een beam search decoder gebruikt met beam width $k = 5$ in model A en greedy decoding ($k = 1$) voor model B. In model C experimenteren we met de impact van de beam width op de WER.

10 Resultaten en analyse

Eerst wordt er geëxperimenteerd met G2P op woordniveau. De beste Word en Phoneme Error Rates van de drie modellen worden samen met de nauwkeurigheid van het huidige (rule-based) systeem van Nuance weergegeven in Tabel 5. Voor alle drie de datasets geeft model C de beste PER en WER van de drie neurale architecturen. De WER van het huidige systeem is zeer laag omdat de evaluatie is uitgevoerd met een woordenlijst die zeer gelijkaardig is aan het lexicon waar het rule-based systeem gebruik van maakt. Wanneer de lexicon lookup wordt uitgeschakeld en het model geforceerd wordt om de ketting linguïstische regels toe te passen, zien we dat de nauwkeurigheid een stuk lager ligt. Deze laatste cijfers kunnen beschouwd worden als de nauwkeurigheid op out-of-vocabulary woorden.
Merk op dat de resultaten tussen de talen onderling niet vergelijkbaar zijn, gezien de grootte van de dataset verschilt en sommige talen meer fonetisch neergeschreven worden dan andere en zo eenvoudiger te converteren zijn.

Tabel 5: Vergelijking tussen de beste resultaten van de verschillende architecturen met het huidige systeem.

	CMUDict		**Turks**		**Kroatisch**	
Model	**PER** (%)	**WER** (%)	**PER** (%)	**WER** (%)	**PER** (%)	**WER** (%)
Rule-based	-	-	1,45	5,50	1,93	5,68
Rule-based*	-	-	12,09	50,09	21,51	85,31
Model A	6,00	24,50	3,31	15,03	6,43	34,29
Model B	7,22	28,76	4,34	19,77	6,89	37,33
Model C	**4,73**	**20,24**	**3,07**	**14,53**	**5,00**	**29,10**

* Enkel fallback-systeem, zonder lexicon

Bij model A wordt er geëxperimenteerd met het aantal lagen l, de dimensie d van de vectoren in de RNN-cel en het type attention. Deze experimenten zijn uitgevoerd op versie 0.7b van CMUDict, waarbij klemtoonsuffixen wel aanwezig zijn. Voor alle modellen worden LSTM-cellen gebruikt, die op een bidirectionele manier gestapeld worden. Het aantal lagen staat hier voor een concatenatie van een forward en backward pass. De PER en WER worden samen met de modelgrootte en inference-tijd weergegeven in Tabel 6. De snelheid wordt gemeten bij *inference* van één woord op CPU en wordt uitgemiddeld over 200 woorden. In de tabel is te zien dat het aantal lagen minder invloed heeft op de performantie dan de dimensie en het type attention. De efficiëntie stijgt met de grootte van de LSTM-units, maar een verdubbeling van het aantal units zorgt echter wel voor een sterke stijging in inference-tijd en modelgrootte.

Tabel 6: De invloed van het aantal lagen l, de grootte d en het type attention op de PER en WER van model A bij training en evaluatie op CMUDict 0.7b.

l	d	**Type attention**	**PER** (%)	**WER** (%)	**Grootte** (MB)	**Snelheid** (ms)
3	256	Geen attn.	9,9	39,4	96,4	39,0
	512		10,8	41,0	24,2	117,1
		Bahdanau	10,1	40,7	98,3	175,6
		Luong	9,4	37,2	146	173,9
2	256		12,7	46,1	24,7	36,4
	512		9,2	37,3	98,4	113,3
	1 024		**9,0**	**36,7**	392	401,5

Bij model B wordt de invloed van het aantal lagen op de PER en WER bekeken. De trainingsfase van model B verloopt naar eigen ervaring zeer instabiel wanneer we hyperparameters gebruiken verschillend van 4 lagen, 256 hidden units en een filterbreedte $k = 3$. Tabel 7 toont de invloed van het aantal lagen l bij $d = 256$. Een model met $d > 256$ slaagt er niet in om te convergeren.

Tabel 7: De invloed van het aantal lagen l op de PER en WER van model B bij training en evaluatie op CMUDict.

l	**PER** (%)	**WER** (%)
3	24,89	59,98
4	**7,22**	**28,76**
5	26,18	67,86

Ook bij model C wordt geëxperimenteerd met een aantal parameters: het aantal lagen l, het aantal hidden units d en het aantal attention heads h. De resultaten voor de Turkse dataset worden in Tabel 8 weergegeven. De kwaliteit van het model stijgt met het aantal trainbare parameters, maar het maximum wordt bereikt bij 5 lagen en 256 units. Het is opmerkelijk dat de modellen met $l = 2$ en $d = 128$ ondanks hun klein aantal parameters nog steeds een aanvaardbare PER en WER halen. Wanneer een model in productie gebruikt moet worden is een zo klein mogelijke modelgrootte vereist, vooral voor applicaties die niet communiceren met een server. Er kan dus een goede trade-off gebeuren tussen de nauwkeurigheid van de conversie en de modelgrootte.

Ook de invloed van de *beam width* k tijdens *inference* wordt geanalyseerd. In de resultaten weergegeven in Tabel 9 zien we een minieme daling in PER en WER ten opzichte van *greedy search* ($k = 1$). Er wordt geen verbetering waargenomen vanaf $k > 2$.

Tabel 8: De invloed van het aantal lagen l, het aantal units d en het aantal attention heads h, de grootte van het model, de snelheid bij inference van één woord en de Error Rates van model C op Turkse data.

l	d	h	**PER** (%)	**WER** (%)	**Grootte** (MB)	**Snelheid** (ms)
2	128	1	4,79	23,29	3,6	41,2
		2	4,29	20,73	3,6	44,1
3	256	4	3,30	15,50	15,1	80,5
	512		16,54	56,69	48,3	160,8
4	256		3,07	14,53	20,2	81,7
5	256		**3,04**	**14,50**	25,2	104,0
	512		43,49	93,25	80,4	-
6	512	8	53,99	96,76	96,4	-

Tabel 9: De invloed van de *beam width* op de PER en WER. Uitgevoerd op model C met de CMUDict dataset.

k	**PER** (%)	**WER** (%)
1	4,91	20,48
2	**4,86**	**20,42**
3	4,87	**20,42**
4	4,87	**20,42**

Tot slot wordt het beste model (model C) vergeleken met eerdere state-of-the-art resultaten op de CMUDict dataset. Tabel 10 laat zien dat de prestaties van het eigen model (laatste regel) zeer dicht liggen bij het recentste werk geleverd in het G2P-domein.

Tabel 10: Vergelijking van voorafgaande resultaten op CMUDict met het beste eigen model.

Model	**PER (%)**	**WER (%)**
Joint n-gram model (Galescu and Allen, 2002)	7,0	28,5
Joint ME n-gram model (Chen, 2003)	5,9	24,7
Joint-sequence model (Bisani and Ney, 2008)	5,88	24,53
LSTM met full-delay (Rao et al., 2015)	9,11	30,1
Encoder-decoder LSTM (Yao and Zweig, 2015)	7,63	28,61
Encoder-decoder LSTM met globale attention (Toshniwal and Livescu, 2016)	5,04	21,69
Ensemble van 5 modellen (Toshniwal and Livescu, 2016)	4,69	20,24
Encoder CNN, decoder BLSTM (Yolchuyeva et al., 2019)	4,81	25,13
Transformer, 4 layers 256 dimensions 8 attn.-heads	**4,73**	**20,24**

10.1 Categorisatie van de gemaakte fouten

Het kan interessant zijn om te weten waar precies de voorspellingen van een neuraal netwerk in de fout gaan. Om dit te achterhalen, bekijken we de fouten zowel op foneemniveau als in functie

van de woordlengte.

Voor het beste model (model C) wordt het type fouten op foneemniveau bekeken en vergeleken met het huidige systeem. Het type fouten kan verkregen worden door in de bewerkingsafstand (Codevoorbeeld 1) een backtrace-matrix toe te voegen, zodat de effectieve bewerking (vervanging, verwijdering of invoeging van een foneem) gekend is. Vervolgens worden de gemaakte fouten opgedeeld in categorieën en wordt hun aandeel in de totale PER berekend. In Tabel 11 en 12 zijn de meest gemaakte fouten voor het Kroatisch en het Turks te zien. Omdat de rule-based benadering met lexicon weinig zinvolle fouten maakt op de gebruikte testdataset, bekijken we enkel de fallback voor out-of-vocabulary woorden.

In het Kroatisch worden de meeste fouten gemaakt op tooncontouren en lengte van de klinkers. Tooncontouren zijn in grotere mate aanwezig in de Kroatische taal in vergelijking met het Turks en zijn moeilijk te beschrijven met linguïstische regels. De spreiding van het type fout is ongeveer gelijk bij de neurale benadering als bij de rule-based strategie, wat impliceert dat het neuraal model de aanwezige regels in de taal correct leert.

Tabel 11: Meeste gemaakte fouten voor model C en de rule-based fallback op de Kroatische dataset.

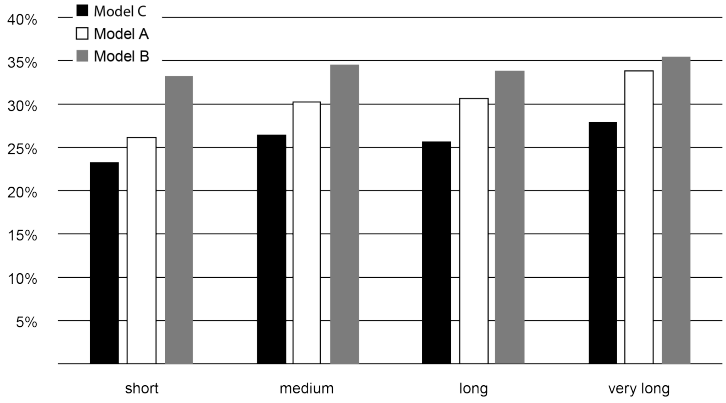
Fout	**Model C** (%)	**Rule-based** (%)
Tooncontouren	1,37	6,36
Klemtoon	1,35	6,36
Klinkerlengte	1,22	3,47
Tooncontouren + Klinkerlengte	0,31	1,42
Andere	0,75	4,19
Totaal (PER)	5,00	21,8

In de Turkse dataset daarentegen maakt de rule-based strategie relatief meer fouten op klemtoon dan een neuraal model. Over- en underprediction zijn een veel voorkomend probleem bij neurale G2P en MT, en treden op wanneer het netwerk er niet in slaagt om de attention te leggen op een karakter, of ze legt het er tweemaal op. Het is opmerkelijk dat de meest gemaakte fouten zowel bij Turks als Kroatisch te maken hebben met suprasegmentalen. De klinkers en medeklinkers worden grotendeels goed voorspeld. Globaal presteert een neuraal netwerk steeds beter dan het rule-based fallback-systeem.

Tabel 12: Meest gemaakte fouten voor model C op de Turkse dataset.

Fout	**Model C** (%)	**Rule-based** (%)
Klemtoon	1,96	8,12
Klinkerlengte	0,34	0,75
Foute klinker	0,19	0,38
Underprediction	0,17	1,11
Foute medeklinker	0,15	0,30
Overprediction	0,12	0,57
Lettergreepgrenzen	0,08	0,41
Andere	0,29	0,49
Totaal (PER)	3,30	12,12

Net zoals Toshniwal and Livescu (2016), delen we de woorden op in vier categorieën op basis van de lengte van de orthografie: kort (*short*, lengte ≤ 6), gemiddeld (*medium*, lengte $\in \{7, 8\}$), lang (*long*, lengte $\in \{9, 10\}$) en zeer lang (*very long*, lengte ≥ 11). Op Figuur 17 is een kleine globale stijging van de WER te zien naarmate de woorden langer worden. Vooral model A kent de grootste stijging. Dit is te verklaren door de bias die RNN's hebben ten opzichte van hun rechtstreekse voorganger (of hun buur in het geval bidirectionele LSTM's), ondanks de toevoeging van memory cellen bij LSTM's. Daardoor presteren ze slechter in het modelleren van long-range dependencies, welke in grotere mate aanwezig zijn in lange woorden. Deze tekortkomingen zijn aangehaald door Lai et al. (2015), Linzen et al. (2016) en Liu et al. (2017).

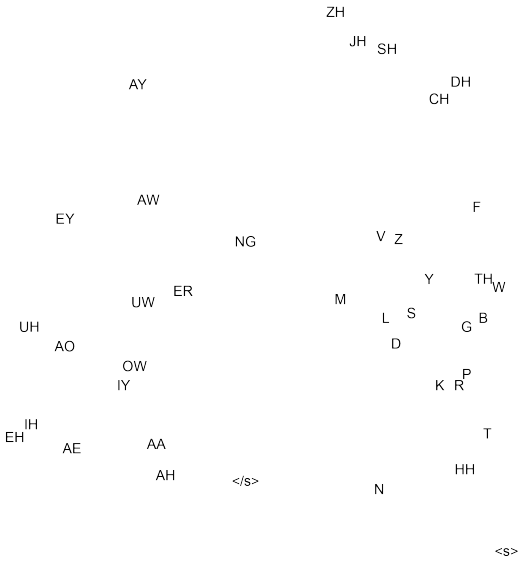


Figuur 17: Vergelijking van WER in functie van de woordlengte voor verschillende modellen.

10.2 Visualisatie character embedding

In Figuur 18 is een visualisatie van de character embedding van het foneemalfabet uit model A te zien. Uit de 256 dimensies zijn er via principale componentenanalyse (PCA) twee dimensies gekozen die de grootste variantie kennen. De spreiding van de verschillende fonemen gedraagt zich als verwacht. De klinkers zijn allemaal links in de embeddingruimte te zien. Rechtsboven zijn de zogenaamde postalveolaire medeklinkers gegroepeerd. Deze medeklinkers komen onder meer voor bij de woorden "ship", "chill", "vision" en "jump". Ook de `DH`-klank uit "father" is aanwezig in deze cluster. Midden rechts bevinden zich de andere medeklinkers. De speciale `<s>`

en </s>-tekens bevinden zich relatief ver van andere fonemen.



Figuur 18: Visualisatie van twee dimensies uit de character embedding via PCA.

10.3 Training op zinsniveau

Vervolgens wordt een Transformermodel getraind op zinsniveau. Omdat we willen dat de Transformer ook verbanden leert over verschillende woorden heen, trainen we met een groter model van 6 lagen, 512 hidden units en 8 attention heads.

In het begin van de trainingsfase ziet het model een zin als één lang woord. Een spatie heeft initieel even weinig betekenis als elk ander grafeem. Het doel is dat het model geleidelijk leert wat de woordgrenzen zijn, dat elk woord exact één primaire klemtoon kent en dat sommige woorden context vereisen uit de rest van de zin.
Na de training op zinsniveau wordt het model gevraagd een reeks zinnen te converteren, die een gekend homograaf bevatten. Er wordt een beam width $k = 2$ gebruikt. De resultaten op die homografen zijn weergegeven in Tabel 13. In de laatste kolom is de gemiddelde nauwkeurigheid per homograaf te zien, welke varieert tussen 50,1% en 94,1%.
Een ander Transformernetwerk ($l = 4$, $d = 256$, $h = 4$) werd op woordniveau getraind op een lexicon van 305 051 Amerikaans Engelse woorden en wist alle homografen uit de tabel correct om te zetten naar een van de twee transcripties. Een ideaal model getraind op woordniveau heeft dus een gemiddelde nauwkeurigheid van 50%.

Met een gemiddelde nauwkeurigheid van 80,8% kunnen we concluderen dat een model getraind op zinsniveau beter presteert in de transcriptie van homografen in vergelijking met een model dat is getraind op woordniveau. Het huidige mechanisme bij Nuance maakt gebruik van een rule-based benadering die onder andere de woordklasse van het homograaf analyseert en op basis daarvan de transcriptie voorspelt. Dit systeem haalt een nauwkeurigheid van meer dan 90% op frequent voorkomende homografen zoals die in Tabel 13. De bestaande rule-based benadering voor Amerikaans Engels wordt dus beter niet vervangen door een neuraal model. Wanneer nog geen rule-based model voor een taal geconstrueerd is en er data beschikbaar is op zinsniveau,

kan een model getraind op zinsniveau ingeschakeld worden.

In kolom 3 is de frequentie van elk homograaf in de corpus te zien. Het valt op dat bij homografen met een nauwkeurigheid minder dan 70% ("lead", "read", "tears") de meest voorkomende uitspraak telkens de hoogste nauwkeurigheid heeft. Dit verschijnsel is te verklaren door een *frequency bias*. Door het combineren van de corpora met de specifiek geselecteerde zinnen is het niet eenvoudig om ervoor te zorgen dat alle uitspraken van een homograaf evenveel aanwezig zijn in de dataset. Bovendien zal een homograaf in de praktijk ook een meer dominante uitspraak hebben.

Tabel 13: Resultaten op homografen van het Transformermodel getraind op zinsniveau.

Orthografie	**Transcriptie**	**Frequentie**	**Accuracy** (%)	**Gemiddeld** (%)
bow	B AW1	999	39,3	68,1
bow	B OW1	1 572	96,9	
content	K AA1 N T EH0 N T	1 908	97,6	85,1
content	K AH0 N T EH1 N T	1 117	72,6	
lead	L EH1 D	1 026	0,9	50,1
lead	L IY1 D	3 032	99,3	
live	L AY1 V	1 362	91,3	90,4
live	L IH V	1 156	89,5	
present	P R EH1 Z AH0 N T	1 283	79,8	87,4
present	P R IY0 Z EH1 N T	896	95,0	
read	R EH1 D	1 844	24,9	62,3
read	R IY1 D	2 123	99,7	
record	R EH1 K ER0 D	1 725	91,6	83,7
record	R IH0 K AO1 R D	411	75,8	
subject	S AH1 B JH IH0 K T	2 269	94,0	90,9
subject	S AH0 B JH EH1 K T	348	87,8	
tear	T EH1 R	1 650	87,8	90,3
tear	T IH1 R	1 748	92,8	
tears	T EH1 R Z	796	44,1	69,5
tears	T IH1 R Z	1 466	95,0	
used (to)	Y UW1 S T	470	93,0	94,1
used	Y UW1 Z D	2 725	95,2	
wind	W AY1 N D	825	89,0	91,2
wind	W IH1 N D	1 335	93,4	
winds	W AY1 N D Z	823	80,9	87,3
winds	W IH1 N D Z	1 270	93,7	
Totaal Gemiddeld (%)				$80,8 \pm 13,7$

Tot slot wordt de globale score van een model getraind op zinsniveau vergeleken met het model getraind op woordniveau. Beide modellen worden geëvalueerd op zowel lexicon- als corpusdata,

en de resultaten worden vergeleken in Tabel 14. Er is duidelijk te zien dat de modellen elk beter presteren op de eigen data, hoewel ze geëvalueerd worden op een ongeziene testset. De grote verschillen in resultaten tussen de verschillende gevallen vallen integraal te verklaren door frequentie.

De corpusdata bevat slechts een kleine subset ($< 20\%$) van alle woorden aanwezig in het lexicon. Een model getraind op zinsniveau zal veel uitzonderingen op regels, eigennamen en woordgroepen met onregelmatige uitspraak ofwel niet gezien hebben, ofwel worden ze verdrongen door woorden die veel voorkomen in de corpus. Hierdoor presteert een model getraind op zinsniveau zeer goed op eigen data, maar slechter op lexicondata.
Een model getraind op woordniveau daarentegen heeft elk woord exact één keer gezien. Onregelmatige woorden die in de realiteit weinig voorkomen hebben een even sterke impact op de gewichten van een netwerk als woorden die bepaalde linguïstische regels volgen.

Wanneer de corpusdata een goede steekproef is van de zinnen die in realiteit moeten geconverteerd worden, kan het interessant zijn om een model op zinsniveau te trainen. Training op zinsniveau betekent bovendien niet per se dat de *inference* ook op zinsniveau moet gebeuren. Een dergelijk model kan evengoed overweg met enkele woorden of met een deelzin.

Tabel 14: Vergelijking van een Transformermodel getraind op woordniveau met een model getraind op zinsniveau bij evaluatie op zowel lexicon- als corpusdata.

	Lexicondata		**Corpusdata**	
Training model	**PER** (%)	**WER** (%)	**PER** (%)	**WER** (%)
Woordniveau	6,27	32,81	18,29	54,89
Zinsniveau	17,33	64,69	2,14	4,71

11 Conclusie

Na grondige studie van en experimentatie met verschillende neurale architecturen, levert de Transformerarchitectuur de beste Word en Phoneme Error Rates op voor verschillende talen. De resultaten sluiten nauw aan bij de huidige state-of-the-art neurale modellen in het G2P-domein met een WER van $20,24\%$ en een PER van $4,73\%$ op de CMUDict dataset. Tot op heden blijken modellen die goed presteren in machine translation ook zeer efficiënt te zijn in grafeem-naar-foneemconversie waarbij alignering van grafeem- en foneemparen niet a priori aanwezig is. Het verband tussen MT en G2P kan dus verder blijven doorgetrokken worden en de nieuwste voorstellen in het MT-domein kunnen telkens in overweging genomen worden voor de G2P-taak.

Alle modellen slagen er goed in om de many-to-many alignering te leren en kiezen met goede betrouwbaarheid de correcte klinkers en medeklinkers. De grootste moeilijkheden waar de modellen mee kampen zijn suprasegmentalen, zoals klemtoon, klinkerlengte en tooncontouren, welke met een rule-based benadering even moeilijk te modelleren zijn. Waar eerder voorgestelde neurale architecturen meer moeite hebben met langere woorden, blijkt de Transformerarchitectuur robuuster te zijn tegen dit probleem.

Wanneer een Transformermodel getraind wordt op zinsniveau haalt het een significant hogere nauwkeurigheid op homografen in vergelijking met een model getraind op woordniveau, maar een rule-based benadering geeft nog steeds de beste resultaten.

Hoewel de huidige rule-based modellen in gebruik bij Nuance een hogere betrouwbaarheid geven, vragen ze linguïstische kennis en een grote opstartkost vooraleer een eerste prototype gebruikt kan worden. Tegenover rule-based modellen halen neurale modellen op zeer korte tijd een goede accuraatheid zonder tussenkomst van een expert. Ze kunnen eenvoudig als prototype voor een nieuwe taal in productie gezet worden, tot wanneer een rule-based systeem de foutpercentages evenaart. Beiden kunnen ook gecombineerd worden om out-of-vocabulary woorden beter af te handelen. De modelgrootte en snelheid van conversie kan verminderd worden met slechts een kleine daling in nauwkeurigheid tot gevolg.

Bibliografie

Alammar, J. (2018). The illustrated transformer. `http://jalammar.github.io/illustrated-transformer`.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bengio, S., Vinyals, O., Jaitly, N., and Shazeer, N. (2015). Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1171–1179.

Bisani, M. and Ney, H. (2008). Joint-sequence models for grapheme-to-phoneme conversion. *Speech communication*, 50(5):434–451.

Chen, S. F. (2003). Conditional and joint models for grapheme-to-phoneme conversion. In *INTERSPEECH*.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, 14(2):179–211.

Elovitz, H., Johnson, R., McHugh, A., and Shore, J. (1976). to-sound rules for automatic translation of english text to phonetics. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 24(6):446–459.

Galescu, L. and Allen, J. F. (2002). Pronunciation of proper names with a joint n-gram model for bi-directional grapheme-to-phoneme conversion. In *Seventh International Conference on Spoken Language Processing*.

Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. (2017). Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1243–1252. JMLR. org.

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE.

Jordan, J. (2018). Setting the learning rate of your neural network. `https://www.jeremyjordan.me/nn-learning-rate/`.

Kalchbrenner, N., Espeholt, L., Simonyan, K., Oord, A. v. d., Graves, A., and Kavukcuoglu, K. (2016). Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*.

Kurita, K. (2017). Paper dissected: "attention is all you need" explained. `http://mlexplained.com/2017/12/29/attention-is-all-you-need-explained`.

Lai, S., Xu, L., Liu, K., and Zhao, J. (2015). Recurrent convolutional neural networks for text classification. In *Twenty-ninth AAAI conference on artificial intelligence*.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710.

Linzen, T., Dupoux, E., and Goldberg, Y. (2016). Assessing the ability of lstms to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4:521–535.

Liu, F., Baldwin, T., and Cohn, T. (2017). Capturing long-range contextual dependencies with memory-enhanced conditional random fields. *arXiv preprint arXiv:1709.03637*.

Luong, M., Brevdo, E., and Zhao, R. (2017). Neural machine translation (seq2seq) tutorial. *https://github.com/tensorflow/nmt*.

Luong, M.-T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

Mousa, A. E.-D. and Schuller, B. W. (2016). Deep bidirectional long short-term memory recurrent neural networks for grapheme-to-phoneme conversion utilizing complex many-to-many alignments. In *Interspeech*, pages 2836–2840.

Neubig, G. (2017). Neural machine translation and sequence-to-sequence models: A tutorial. *arXiv preprint arXiv:1703.01619*.

Ping, W., Peng, K., Gibiansky, A., Arik, S. Ö., Kannan, A., Narang, S., Raiman, J., and Miller, J. (2017). Deep voice 3: 2000-speaker neural text-to-speech. *CoRR*, abs/1710.07654.

Pitts, W. (1942). Some observations on the simple neuron circuit. *The bulletin of mathematical biophysics*, 4(3):121–129.

Rao, K., Peng, F., Sak, H., and Beaufays, F. (2015). Grapheme-to-phoneme conversion using long short-term memory recurrent neural networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4225–4229. IEEE.

Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Shen, J., Pang, R., Weiss, R. J., Schuster, M., Jaitly, N., Yang, Z., Chen, Z., Zhang, Y., Wang, Y., Skerrv-Ryan, R., et al. (2018). Natural tts synthesis by conditioning wavenet on mel spectrogram predictions. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4779–4783. IEEE.

Sotelo, J., Mehri, S., Kumar, K., Santos, J. F., Kastner, K., Courville, A. C., and Bengio, Y. (2017). Char2wav: End-to-end speech synthesis. In *ICLR*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Toshniwal, S. and Livescu, K. (2016). Jointly learning to align and convert graphemes to phonemes with neural attention models. In *2016 IEEE Spoken Language Technology Workshop (SLT)*, pages 76–82. IEEE.

Van Den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. W., and Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. *SSW*, 125.

Vaswani, A., Bengio, S., Brevdo, E., Chollet, F., Gomez, A. N., Gouws, S., Jones, L., Kaiser, Ł., Kalchbrenner, N., Parmar, N., et al. (2018). Tensor2tensor for neural machine translation. *arXiv preprint arXiv:1803.07416*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Wang, W., Xu, S., and Xu, B. (2016). First step towards end-to-end parametric tts synthesis: Generating spectral parameters with neural attention. In *INTERSPEECH*.

Wang, Y., Skerry-Ryan, R., Stanton, D., Wu, Y., Weiss, R. J., Jaitly, N., Yang, Z., Xiao, Y., Chen, Z., Bengio, S., et al. (2017). Tacotron: A fully end-to-end text-to-speech synthesis model. *arXiv preprint arXiv:1703.10135*.

Yao, K. and Zweig, G. (2015). Sequence-to-sequence neural net models for grapheme-to-phoneme conversion. *arXiv preprint arXiv:1506.00196*.

Yolchuyeva, S., Németh, G., and Gyires-Tóth, B. (2019). Grapheme-to-phoneme conversion with convolutional neural networks. *Applied Sciences*, 9(6):1143.

Bijlage A Datavoorbereiding

```
# 1. Read config file from CLI
parser = argparse.ArgumentParser()
parser.add_argument(
    'config_file',
    type=argparse.FileType('r'),
    help='configuration file, see default.json for options')
args = parser.parse_args()

# 2. Load default and user data
cfg = json.load(open('default.json'))
cfg.update(json.load(args.config_file))
cfg = munchify(cfg)

# 3. Tokenize transcription
symbol_table = SymbolTable(cfg.symbol_table, cfg.language)
output_pairs = []
grapheme_vocab = set()
for line in open(cfg.transcription_file, encoding='utf-8'):
    columns = line.split(cfg.separator)

    orthography = columns[cfg.orthography_column].strip()
    transcription = columns[cfg.transcription_column].strip()

    if transcription and orthography:
        orthography = orthography.replace(
            ' ', cfg.replace_orthography_spaces)
        grapheme_vocab.update(orthography)

        success, string = symbol_table.tokenize_transcription(
            transcription)

        if success:
            output_pairs.append(orthography + '\t' + ''.join(string))
        else:
            logging.error('%s (%s)', string[0], string[1])

# 4. Write vocab files
utils.write_vocab(
    grapheme_vocab,
    cfg.extra_grapheme_tokens,
    os.path.join(cfg.output_directory, 'vocab.graph'))

# 5. Remove duplicate lines
if cfg.remove_duplicates:
    orig_len = len(output_pairs)
    output_pairs = np.unique(output_pairs)
    new_len = len(output_pairs)
```

```
    logging.info('Duplicates removed: %d', orig_len - new_len)

# 6. Write full dataset
full = open(
    os.path.join(cfg.output_directory, 'full.txt'),
    'w+',
    encoding='utf-8')
full.writelines('\n'.join(output_pairs))

# 7. Write dataset splits
train, val = train_test_split(
    output_pairs,
    test_size=cfg.split[1],
    random_state=1)
train, test = train_test_split(
    train,
    test_size=(cfg.split[2] / (cfg.split[0] + cfg.split[2])),
    random_state=1)

utils.write_split(train, os.path.join(cfg.output_directory, 'train'))
utils.write_split(val, os.path.join(cfg.output_directory, 'val'))
utils.write_split(test, os.path.join(cfg.output_directory, 'test'))
```

Bijlage B Levenshteinafstand met backtrace

```
def levenshtein_backtrace(s, t):
    if s == t:
        return 0, []

    v0 = np.arange(len(t) + 1)
    v1 = np.empty(len(t) + 1)
    B = np.zeros((len(t) + 1, len(s) + 1), dtype=np.int8)
    B[1:, 0] = 0
    B[0, 1:] = 1

    for i in range(len(s)):
        v1[0] = i + 1
        for j in range(len(t)):
            operations = [v1[j] + 1,        # deletion
                          v0[j + 1] + 1,    # insertion
                          v0[j] + (0 if s[i] == t[j] else 1)]
                          # substitution or correct
            b = np.argmin(operations)
            B[j+1, i+1] = b
            v1[j + 1] = operations[b]
        v0 = np.copy(v1)

    i, j = B.shape[0]-1, B.shape[1]-1
    mistakes = []
    while (i, j) != (0, 0):
        if B[i, j] == 2:
            if t[i-1] != s[j-1]:
                mistakes.append('{} -> {}'.format(s[j-1], t[i-1]))
            i, j = i-1, j-1
        elif B[i, j] == 0:
            mistakes.append('extra {}'.format(t[i-1]))
            i, j = i-1, j
        elif B[i, j] == 1:
            mistakes.append('missed {}'.format(s[j-1]))
            i, j = i, j-1

    return v1[-1], mistakes
```

Bijlage C Berekening van de WER / PER met tolerantie op homografen

```
def wer_per(config):
    full = dict()
    for line in open(config.full_dataset, encoding='utf-8'):
        grapheme, phoneme = line.split('\t', 2)
        grapheme = grapheme.strip()
        phoneme = phoneme.strip()
        if grapheme in full:
            full[grapheme].append(phoneme)
        else:
            full[grapheme] = [phoneme]

    input_file = open(config.input, encoding='utf-8')
    pred_file = open(config.prediction, encoding='utf-8')
    word_count, correct_words = 0, 0
    phoneme_count, total_dist = 0, 0
    for src in input_file:
        y_pred = pred_file.readline().strip()
        src = src.strip()

        homographs = full[src]

        dists = [
            levenshtein(y.split(' '), y_pred.split(' '))
            for y in homographs]
        min_dist = np.argmin(dists)
        total_dist += dists[min_dist]
        phoneme_count += len(homographs[min_dist].split(' '))

        word_count += 1
        if y_pred in homographs:
            correct_words += 1

    wer = 1 - (correct_words / word_count)
    per = total_dist / phoneme_count

    return wer, per
```