

NOTAM Smartification

VM 01
MASTER OF SCIENCE IN ENGINEERING

presented by
MARC BRAVIN

Lucerne School of Information Technology
Lucerne University of Applied Sciences and Arts
6343 Rotkreuz, Switzerland

January 31, 2020

Advisor	Prof. Dr. Marc Pouly Lucerne University of Applied Sciences and Arts, 6343 Rotkreuz, Switzerland marc.pouly@hslu.ch
Second supervisor	Daniel Pfäffli Lucerne University of Applied Sciences and Arts, 6343 Rotkreuz, Switzerland daniel.pfaeffli@hslu.ch
Expert	Dr. Johannes Huber SAFEmine Ltd, 6300 Zug, Switzerland johannes.huber@hexagon.com

I hereby declare that I have prepared the present work independently and have used nothing other than the specified aids. All text sections, citations or contents of other authors used have been explicitly marked as such.

A handwritten signature in black ink, appearing to read 'Marc Bravin', with a stylized, cursive script.

Rotkreuz, 6th January 2020, Marc Bravin

Abstract

Notice to airmen (NOTAM) are text messages targeted to pilots to make them aware of short term events and obstacles. Typical examples are military actions in airspaces or closed runways. Pilots in Switzerland are obliged to conduct a NOTAM briefing before take-off. Thousands of NOTAMs are broadcasted every day and poorly filtered concerning a pilot’s envisaged route and contain a lot of irrelevant information. Smart NOTAM is a service offered by Skyguide that filters irrelevant messages for a specific route or period. This service also includes a smartification process that shortens the messages by omitting meaningless words and phrases. It also converts the messages into a standardised phraseology.

This project provides an overview of current natural language processing literature in general and machine translation, as well as text summarization in the domain of NOTAM messages. Having access to thousands of NOTAMs with their smartified version, a machine learning model that performs the smartification process was implemented and evaluated.

A data quality assessment showed that from 1.1 million messages, almost 50% of the raw NOTAM messages are already smart and thus do not need to be smartified at all. Furthermore, raw messages leave the smartification process either unchanged or with substantial changes. Based on these findings, an additional model was implemented which classifies whether a NOTAM needs to be smartified or not.

For the classification task, several text classification models based on either recurrent neural networks, convolutional neural networks (CNNs) or transformers were evaluated. The text classification model based on CNNs proved to be the most accurate and achieved an F_1 score of 93.66% on unseen test data. To perform the smartification process, a sequence to sequence transformer model was trained and evaluated. NOTAMs contain a high number of factual details such as coordinates or frequencies, making the smartification process vulnerable to the inaccurate generation of such details. To improve the reproduction of factual details, the transformer model was extended by a pointer-network that allows it to copy words directly from the source text. This way, the evaluated model achieved a $BLEU_4$ score of 86.12% and a $ROUGE_2$ score of 90.23% on unseen test data. Yet there are some cases where the model fails to smartify the messages: if some factual details are added from another source or if the messages are written in a language other than using English abbreviations.

Contents

1	Introduction	1
1.1	Notice to Airmen	1
1.1.1	NOTAM Format	1
1.1.2	Smart NOTAM	2
1.2	Pre-project	2
1.3	Research Questions and Expected Results	3
1.4	Organization of this Report	3
1.5	Acknowledgements	3
2	Related Work	4
2.1	Artificial Neural Networks	4
2.1.1	Perceptron	4
2.1.2	Activation Functions	6
2.1.3	Cost Functions	10
2.1.4	Gradient Descent Algorithm	10
2.1.5	Stabilizing Gradients	12
2.2	Recurrent Neural Networks	14
2.3	Convolutional Neural Networks	17
2.3.1	Convolutional Layers	17
2.3.2	Pooling Layers	18
2.4	Word Embeddings	19
2.4.1	Term Frequency-Inverse Document Frequency	19
2.4.2	Word2Vec	19
2.4.3	GloVe	20
2.4.4	FastText	20
2.4.5	Flair	21
2.4.6	BERT	21
2.5	Text Classification	22
2.5.1	Rule-based Systems	22
2.5.2	Naïve Bayes Classifier	22
2.5.3	Neural Approaches	23
2.5.4	Metrics	26
2.6	Sequence to Sequence Models	28
2.6.1	Recurrent Sequence to Sequence Models	28
2.6.2	Recurrent Sequence to Sequence Models with Attention	29
2.6.3	Convolutional Sequence to Sequence Models	34
2.6.4	Transformer Sequence to Sequence Models	35
2.6.5	Handle Out-of-Vocabulary Words	38
2.6.6	Beam Search	39
2.6.7	Metrics	41
2.6.8	Conclusion	43
3	Setup and Models	44
3.1	NOTAM Dataset	44
3.1.1	Data Quality Assessment	44
3.1.2	Preprocessing	48
3.2	Classification Models	51
3.2.1	Data Preprocessing	51
3.2.2	Metrics	51
3.2.3	Baseline Classification Model	51

3.2.4	Recurrent Classification Model	52
3.2.5	Convolutional Classification Model	53
3.2.6	Combination of CNN and RNN	54
3.2.7	Transformer Classification Model	55
3.2.8	Optimizer Settings	56
3.3	Sequence to Sequence Models	57
3.3.1	Metrics	57
3.3.2	Baseline Seq2Seq Model	58
3.3.3	Transformer Seq2Seq Model	58
3.3.4	Hierarchical Model	62
4	Results	63
4.1	NOTAM Classification	63
4.1.1	Model Comparison	63
4.1.2	Final Evaluation	65
4.2	NOTAM Smartification	67
4.2.1	Model Comparison	67
4.2.2	Hierarchical Model	70
4.2.3	Final Evaluation	71
5	Discussion	73
5.1	Outlook	73
5.1.1	Possible Improvements	73
5.1.2	Extend Smart NOTAM Workflow	74
5.1.3	NOTAM Translation	74
5.2	Conclusion	74
A	Appendix	75
A.1	Demonstration App	75
A.2	Code	76
A.2.1	Jupyter Notebooks	76
A.2.2	NOTAM Classification Model	77
A.2.3	NOTAM Smartification Model	77

Chapter 1

Introduction

This chapter provides an introduction to the topic of notice to airmen and explains the motivation behind this project.

1.1 Notice to Airmen

Notice to airmen (NOTAM) are short text messages targeted to pilots to make them aware of short term events and obstacles. Typical examples are military actions in airspaces, closed runways due to lawn tractors or cranes near airports. In Switzerland, pilots are obliged to conduct a NOTAM briefing before take-off. For historical reasons, NOTAMs are written in a strongly compressed format and vocabulary, making the information hard to access for non-trained readers. In addition, thousands of messages are broadcasted every day, which are poorly filtered concerning a pilot's envisaged route. Despite the strong compression, NOTAMs can still contain a lot of irrelevant information. For example, the fact that a runway is being closed is an important message for a pilot. Whether the reason is a lawn tractor, a pothole or a swarm of birds is irrelevant.

1.1.1 NOTAM Format

NOTAMs are published using all upper case letters, which makes them hard to read. They contain a validity period, geographic area (aerodrome, airstrip, region), flying attitude and a description. These messages are sometimes short-term and may be changed for a specific duration. With the Skyguide portal, a user can view NOTAMs by providing information about departure and destination, duration of flight and date. NOTAMs contain the properties A-G and Q:

A: The affected aerodrome or flight information region (FIR).

B: Contain the start date and time.

C: Contains the end date and time.

D: A specific validity date if the hours are less than 24 hours. E.g. even though a parachute dropping exercise can last multiple days, the hours are often less than 24 hours a day.

E: Actual NOTAM text. It is a description in a strongly compressed format which describes the danger.

F and G: Flight attitude but in any kind of formats.

Q: The Q-code represents the NOTAM in a compressed form. In general, the NOTAM text has more information than the Q-code.

1.1.2 Smart NOTAM

Smart NOTAM¹ is a service offered by Skyguide (for an annual fee) that filters irrelevant messages for a specific route or period. For example, bombing in Syria is unfortunate but irrelevant for a flight from Zurich to Paris. The service further includes a so-called smartification process where the NOTAM property E text is revised to improve its readability and shortened to include only content of operational relevance. Aeronautical information management (AIM) officers from Skyguide perform this smartification process by hand. Table 1.1 shows some examples of the NOTAM text paired with their smartification.

NOTAM (Property E)	SmartNOTAM (Property E)
POSSIBLE DELAYS IN ARR/DEP DUE TO AIR EXER	EXP DLA.
AIRPORT CLOSED DUE SNOW CLEARING.	AP CLSD
LOCALIZER UNSERVICEABLE STOP	RWY 23 ILS LOC U/S.

Table 1.1: Examples of NOTAMs with their smartified version

Another feature of the smart NOTAM service is the correction of the Q-code if it is not correct. The workflow of the NOTAM smartification service is depicted in figure 1.1 and is executed as follows:

1. The first step is to predict whether the Q-code is correct or it has to be changed
2. If the Q-code needs to be changed, the second step is to adapt the Q-code and add Skyguide-specific appendix
3. The third step is to predict whether the NOTAM text (property E) needs to be changed
4. If the NOTAM text needs to be changed, the last step is to smartify the NOTAM text by omitting irrelevant information and converting it into a standardized phraseology.

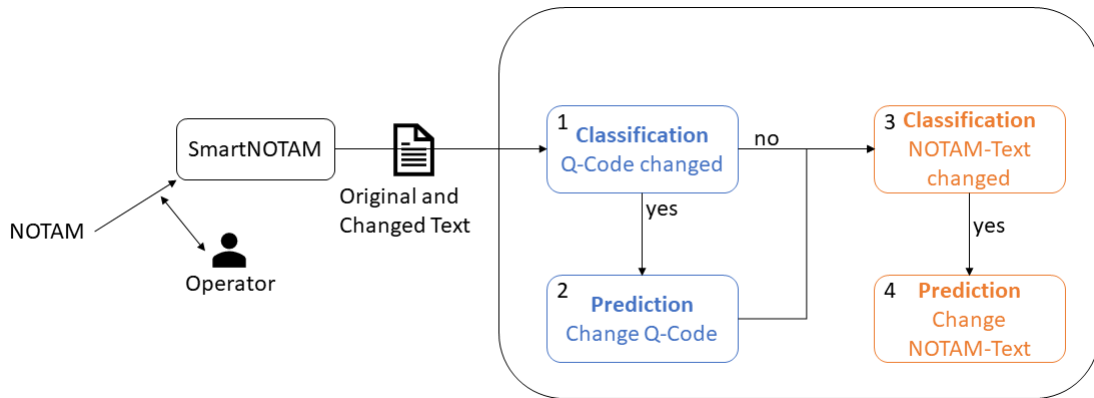


Figure 1.1: The workflow and project scope of the smart NOTAMs

1.2 Pre-project

Prior to this project, a preliminary project was carried out by members of the algorithmic business (ABIZ) research team of the Lucerne University of Applied Sciences and Arts (HSLU). The goal of the project was to classify whether the Q-code is correct or needs to be adapted. The researchers showed that they succeeded in training a machine learning model that achieves an accuracy of around 96% on unseen test data.

¹https://www.skyguide.ch/wp-content/uploads/2019/01/Smart-NOTAM_2018.pdf, accessed 10.10.2019

1.3 Research Questions and Expected Results

This project is intended to provide an overview of current natural language processing (NLP) literature in general and machine translation as well as text summarization in the domain of NOTAM messages. Having access to several hundred-thousands of NOTAM messages paired with their smartified version, a machine learning model should be evaluated to automate the smartification process. The smartification service of Skyguide includes corrections in all NOTAM properties A-G and Q. In the scope of this project, only the smartification of property E (the actual NOTAM text) is considered. In relation to the workflow in figure 1.1, the steps 3) and 4) should be automated.

1.4 Organization of this Report

This report is structured into five chapters. The first chapter gives an overview of the motivation, research questions and expected results of the project. The second chapter describes the concepts used throughout this project. The third chapter introduces the setup and the machine learning models that were implemented to address the research questions. The fourth chapter describes the results of the conducted experiments. Finally, the last chapter discusses the results and describes possibilities to further improve the results.

Notation

Throughout this report, vectors and matrices are written in bold. The most frequently used mathematical notations are listed below.

- Multiplication
- \odot Element-wise multiplication
- θ Parameters of the machine learning model
- \mathbf{x} Vector of inputs of the model
- \mathbf{y} Vector of the ground truth values
- $\hat{\mathbf{y}}$ Vector of the model predictions
- J Cost or loss function
- \mathbf{W} Matrix consisting of the weight vectors for the model
- \mathbf{b} Bias vector of the model
- \mathbf{h} Hidden state of a recurrent neural network

1.5 Acknowledgements

I would like to thank Prof. Dr. Marc Pouly from the Lucerne University of Applied Sciences and Arts, who supported me throughout this project and gave valuable feedback and suggestions.

Equally, I thank Daniel Pfäffli from the ABIZ research team at the Lucerne University of Applied Sciences and Arts, who gave me valuable advice when implementing the machine learning models and presented me the results from the pre-project.

I thank Prof. Martin Jud from the Lucerne University of Applied Sciences and Arts who gave me an introduction into the NOTAM briefing that pilots are obliged to conduct before take-off.

Chapter 2

Related Work

This chapter introduces the concepts used throughout this report. It starts with the basics of neural networks and explains how recurrent neural networks and convolutional neural networks work. Additionally, word embeddings and several methods for text classifications are presented. Finally, the state-of-the-art of sequence to sequence models, which are used to tackle several NLP tasks, is reviewed and presented.

2.1 Artificial Neural Networks

This section introduces the concepts of artificial neural networks (ANNs) which were used to implement the models described in chapter 3.

2.1.1 Perceptron

The evolution of ANNs began with the paper from McCulloch and Pitts (1943). Their work was inspired by biological neuron in the brains of multicellular organisms such as humans. They describe the brain as a net of neurons where the neurons send an impulse if they receive a sufficient number of signals from other neurons within a few milliseconds. Based on these findings, they formulated the McCulloch-Pitts (MP) neuron, a mathematical model that imitates the functionality of a biological neuron. The neuron accepts multiple binary inputs, aggregates them and produces a binary output based on a certain threshold value. Equation 2.1 states how an output \hat{y} is calculated with a single MP neuron where n is the number of inputs and $g(z)$ the threshold function.

$$\hat{y} = g\left(\sum_{k=1}^n x_k\right) \text{ for } x_k \in 0, 1$$
$$g(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{if } z < \theta \end{cases} \quad (2.1)$$

Figure 2.1 shows a schematic representation of a MP neuron with $n = 3$ inputs. The inputs x_1 , x_2 and x_3 are summed up and fed into the threshold function $g(z)$ to produce the model output.

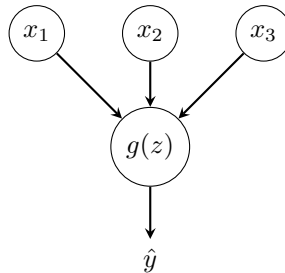


Figure 2.1: A graphical illustration of a MP neuron proposed by McCulloch and Pitts (1943)

Rosenblatt (1958) introduced the perceptron neuron or linear threshold unit (LTU). The perceptron can be viewed as a modification of the MP neuron where the input can be any real number. As shown in equation 2.2, the inputs x_k are weighted by a factor w_k and aggregated. Additionally, a bias b is added and the result is fed into the Heaviside function $h(z)$. The Heaviside function represents a non-linear function that maps negative inputs to zero and inputs larger or equal to zero to one.

$$\hat{y} = h \left(\sum_{k=1}^n w_k x_k + b \right) \text{ for } x_k, w_k, b \in \mathbb{R}$$

$$h(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (2.2)$$

Figure 2.2 shows an example of a perceptron with $n = 3$ inputs. Each input is weighted by a corresponding weight factor. The +1 node represents the bias unit with constant value 1 which is multiplied by the bias b .

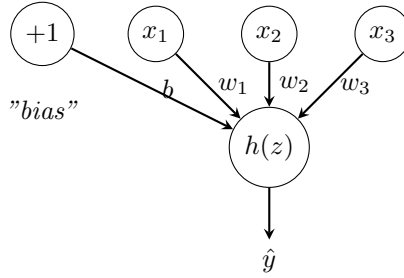


Figure 2.2: A graphical illustration of a perceptron proposed by Rosenblatt (1958)

Multi-Layer Perceptron

Single-layer perceptrons can only be used for linearly separable binary classification problems. The XOR function for example, cannot be solved by a single-layer perceptron. A multi-layer perceptron (MLP) has at least three layers: an input layer, one or more hidden layers and an output layer. Instead of the Heaviside function $h(z)$, other non-linear activation functions $g(z)$ can be applied after each layer.

Equation 2.3 states how the activations $\mathbf{a}^{[l]}$ for a layer l are computed by using the output of the previous layer $\mathbf{a}^{[l-1]}$, where n_l denotes the number of inputs to the layer. For the first layer $l = 1$, the incoming activations are equal to the inputs \mathbf{x} of the network. The outputs of the last layer correspond to the predictions $\hat{\mathbf{y}}$.

$$a_i^{[l]} = g^{[l]} \left(\sum_{k=1}^{n_{l-1}} w_{i,k}^{[l]} \cdot a_k^{[l-1]} + b_i^{[l]} \right) \quad (2.3)$$

Figure 2.3 shows an example of a MLP with one hidden layer and two inputs x_1 and x_2 which produces a single output \hat{y} .

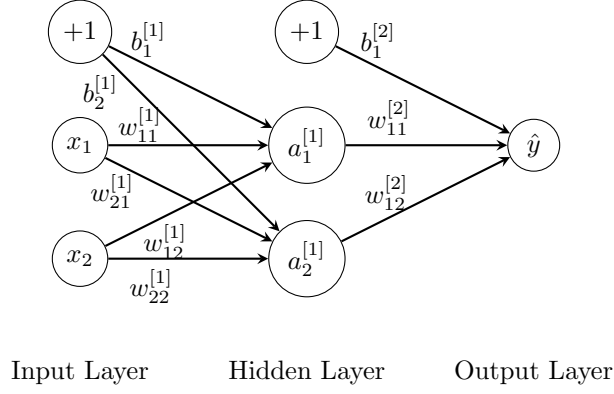


Figure 2.3: An example of a MLP with one hidden layer and two inputs

2.1.2 Activation Functions

Many activation functions exist that can be used instead of the Heaviside function. However, there is no universal activation function that is suitable for every use-case. Each has certain advantages and disadvantages.

Sigmoid Function

The sigmoid function $\sigma(z)$ in equation 2.4 and shown in figure 2.4 maps an input to a value between 0 and 1. Due to its probabilistic interpretation, it is often used in the last layer for a binary classification problem. A major disadvantage of the sigmoid function is that it has saturation regions which can lead to vanishing gradients.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (2.4)$$

The derivative $\sigma'(z)$ of the sigmoid function is given in equation 2.5.

$$\sigma'(z) = (1 - \sigma(z)) \cdot \sigma(z) \quad (2.5)$$

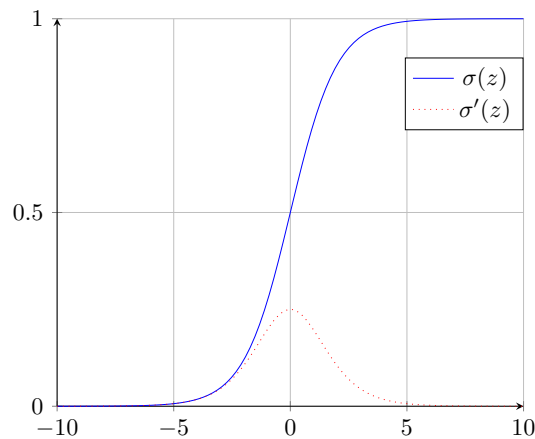


Figure 2.4: The sigmoid activation function

Tangens Hyperbolicus Function

The tangens hyperbolicus (\tanh) function in equation 2.6 returns a value between -1 and 1 . It is closely related to the sigmoid function and also suffers from the vanishing gradient problem.

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2 \cdot \sigma(2z) - 1 \quad (2.6)$$

Equation 2.7 shows that the derivative can be expressed as a combination of the tanh function.

$$\tanh'(z) = 1 - \tanh^2(z) \quad (2.7)$$

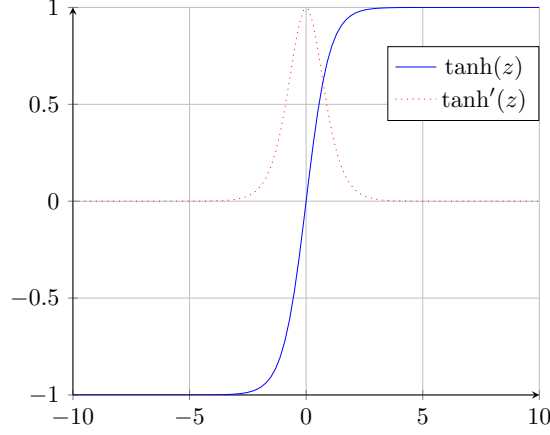


Figure 2.5: The tanh activation function

Rectified Linear Unit Function

The rectified linear unit (ReLU) activation function has been introduced by Glorot et al. (2011). As shown in equation 2.8 and figure 2.6 it is defined as the maximum between the input z and zero. Equation 2.9 shows that its derivative is undefined at zero. The idea behind the ReLU activation function is that it should alleviate the vanishing gradients problem. However, it introduces the dying units problem because the gradient is zero for negative values.

$$\text{ReLU} = \max(0, z) \quad (2.8)$$

$$\text{ReLU}'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases} \quad (2.9)$$

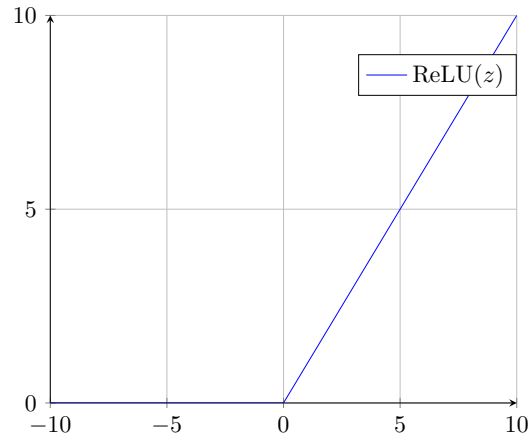


Figure 2.6: The ReLU activation function

Leaky Rectified Linear Unit

The leaky rectified linear unit (LReLU) function is an adaption of the ReLU function and has been proposed by Maas (2013). It alleviates both, the vanishing gradient and the dying unit problem by using a small hyper-parameter α which makes sure that the unit never dies. The gradient is non-zero over the entire domain, unlike the standard ReLU activation function. The equations are shown in 2.10 and 2.11. The function is plotted in figure 2.7 with $\alpha = 0.1$.

$$\text{LReLU}_\alpha = \max(\alpha \cdot z, z) \quad (2.10)$$

$$\text{LReLU}'_\alpha(z) = \begin{cases} 1 & z > 0 \\ \alpha & z < 0 \end{cases} \quad (2.11)$$

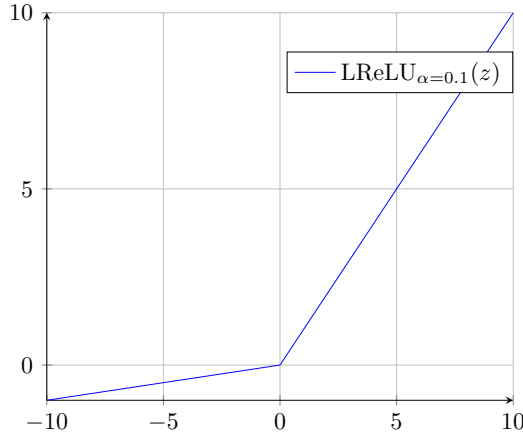


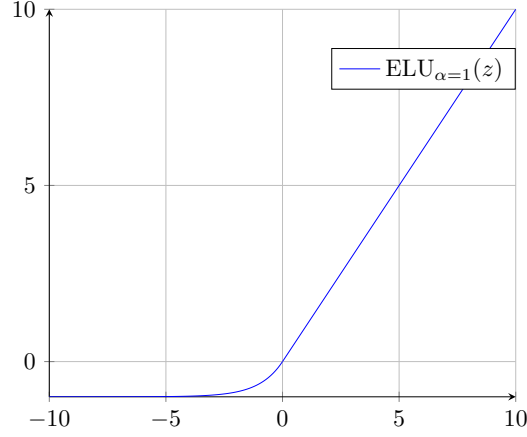
Figure 2.7: The leaky ReLU activation function with $\alpha = 0.1$

Exponential Linear Unit

The exponential linear unit (ELU) activation function is an extension of the LReLU function that also has the property of alleviating the dying units problem. It was first introduced by Clevert et al. (2015). Equations 2.12 and 2.13 denote the formula for the ELU function and its derivative respectively, where α should take a positive value. Figure 2.8 shows a plot of the function with $\alpha = 1$.

$$\text{ELU}_\alpha(z) = \begin{cases} z & z > 0 \\ \alpha \cdot (e^z - 1) & z \leq 0 \end{cases} \quad (2.12)$$

$$\text{ELU}'_\alpha(z) = \begin{cases} 1 & z > 0 \\ \alpha \cdot e^z & z \leq 0 \end{cases} \quad (2.13)$$


 Figure 2.8: The ELU activation function with $\alpha = 1$

Gaussian Error Linear Unit

The Gaussian error linear unit (GELU) function has been proposed by Hendrycks and Gimpel (2016) and is currently the state-of-the-art for transformer models (see section 2.6.4). The authors claim that the increased curvature and non-monotonicity may allow the GELU activation function to approximate complex functions easier. The activation function is calculated by the cumulative distribution function (CDF) of a standard Gaussian distribution $\mathcal{N}(0,1)$ scaled by the input z . The formula is denoted in equation 2.14 and can be approximated using a tanh or sigmoid σ function. Figure 2.9 shows a plot of the function.

$$\text{GELU}(z) = zP(Z \leq z) = z\Phi(z) \approx 0.5z \left(1 + \tanh \left(\sqrt{2/\pi} (z + 0.044715z^3) \right) \right) \approx z \cdot \sigma(1.702z) \quad (2.14)$$

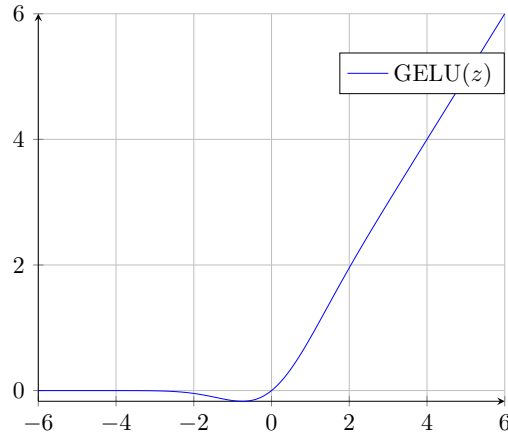


Figure 2.9: The GELU activation function

Softmax Function

The sigmoid function is used for binary classification. If a neural network with K classes is trained, the activation function for the output layer is a softmax function. The output layer consists of K output neurons that predict the probability for the input to be classified to each class. As shown in equation 2.15, the softmax function returns normalized probabilities for the different classes j .

$$\text{softmax}_j(z) = \frac{\exp(z_j)}{\sum_{k=1}^K \exp(z_k)} \quad (2.15)$$

2.1.3 Cost Functions

The goal of training a neural network is to optimize the parameters θ so that the difference between the ground truth y and the prediction \hat{y} is minimal. To measure that difference, a cost function or commonly called loss function needs to be defined.

Mean Squared Error Cost Function

The mean squared error (MSE) cost function J_{MSE} is typically used for regression problem. It calculates the sum of the squared difference of the predictions $\hat{\mathbf{y}}$ and the ground truth values \mathbf{y} and divides it by two times the number of samples m as shown in equation 2.16. The factor 2 is commonly used to simplify the calculation of the derivatives.

$$J_{\text{MSE}}(\theta) = \frac{1}{2m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2 \quad (2.16)$$

Cross Entropy Cost Function

The cross entropy cost function is mostly used for classification problems. Equation 2.17 states how the cost function is computed where m is the number of samples and K the number of classes.

$$J_{\text{CE}}(\theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \cdot \log(\hat{y}_k^{(i)}) \quad (2.17)$$

In case of a binary classification problem, the cross entropy cost function can be written as shown in equation 2.18.

$$J_{\text{BCE}}(\theta) = -\frac{1}{m} \sum_{i=1}^m \left(y^{(i)} \cdot \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \cdot \log(1 - \hat{y}^{(i)}) \right) \quad (2.18)$$

2.1.4 Gradient Descent Algorithm

The most used algorithm to optimize the parameters of a neural network is the gradient descent algorithm. The prerequisite when using the gradient descent algorithm is that the cost function J and all activation functions of the neural network need to be differentiable. Algorithm 1 shows how the algorithm works. The idea behind the algorithm is that the gradient $\Delta J(\theta) = \frac{\partial J}{\partial \theta}$ of the cost function J always points in the direction of the steepest ascent. Thus the negative gradient points in the direction of the steepest descent. At each step the parameters θ are updated by subtracting the gradient multiplied by the learning rate α . The learning rate α is a hyperparameter that controls the magnitude of the step. This is repeated until it converges, i.e. the changes in the parameter fall below a certain threshold.

Algorithm 1 Gradient descent algorithm

initialize parameters θ_t

repeat

$\mathbf{g} \leftarrow \Delta J(\theta_t)$	<i>// Compute the gradient</i>
$\theta_{t+1} \leftarrow \theta_t - \alpha \cdot \mathbf{g}$	<i>// Step into the negative direction of the gradient</i>

until *convergence*

Several adaptations of the gradient descent algorithm exists and are explained in the remainder of this section.

Momentum Algorithm

The main disadvantage of the gradient descent algorithm is that it is not guaranteed to converge in a global optimum. The learning process can get stuck in flat regions where the gradient is small or zero. The momentum approach is a modification of the gradient descent algorithm discussed in Botev et al. (2016) with the goal to overcome flat regions. With the momentum approach, shown in algorithm 2, not only the gradient of the current step is considered, but also the past gradient. The variable \mathbf{m} is computed each step as an exponential moving average between the current and the past gradient. The hyperparameter β_1 controls the decay and the friction of the momentum method and is usually set to 0.9.

Algorithm 2 Momentum algorithm

initialize parameters $\boldsymbol{\theta}_t, \mathbf{m}_t$

repeat

$$\mathbf{m}_{t+1} \leftarrow \beta_1 \cdot \mathbf{m}_t + \alpha \cdot \Delta J(\theta_t)$$
$$\theta_{t+1} \leftarrow \theta_t - \mathbf{m}_{t+1}$$

until *convergence*

Root Mean Square Propagation Algorithm

Choosing an appropriate learning rate α is crucial when applying the gradient descent algorithm. If α is too small, it takes very long to converge. If α is too large, it is very likely that it overshoots the minimum. The root mean square propagation (RMSProp) algorithm, proposed by Tieleman (2012), adaptively adjusts the learning rate. In the direction of slow progress the learning rate is increased, whereas in the direction of fast progress the learning rate is decreased. Algorithm 3 builds an exponentially decaying measure with the typical scale of the components of the gradient vectors \mathbf{s} . Each component of the gradient is then divided by the scale \mathbf{s} . The variable ϵ is an error term that prevents division by zero and is typically a very small number such as $\epsilon = 10^{-8}$. A common default value for the hyperparameter β_2 is 0.999.

Algorithm 3 RMSProp algorithm

initialize parameters θ_t, \mathbf{s}_t

repeat

$$\mathbf{s}_{t+1} \leftarrow \beta_2 \cdot \mathbf{s}_t + (1 - \beta_2) \cdot \Delta J(\boldsymbol{\theta}_t) \odot \Delta J(\boldsymbol{\theta}_t)$$
$$\theta_{t+1} \leftarrow \theta_t - \frac{\alpha}{\sqrt{s_{t+1} + \epsilon}} \odot \Delta J(\theta_t)$$
until *convergence*

Adaptive Moment Estimation Algorithm

The adaptive moment estimation (adam) optimizer has been introduced by Kingma and Ba (2014) and is a combination of the momentum and RMSProp algorithms. It takes advantage of momentum by using a moving average of the gradient and uses the squared gradients to scale the learning rate like RMSProp. Algorithm 4 shows how the momentum \mathbf{m} and the value \mathbf{s} is calculated and then used to update the parameters $\boldsymbol{\theta}$.

Algorithm 4 Adam algorithm

initialize parameters $\theta_t, \mathbf{m}_t, \mathbf{s}_t$

repeat

$$\mathbf{m}_{t+1} \leftarrow \beta_1 \cdot \mathbf{m}_t + (1 - \beta_1) \cdot \Delta J(\boldsymbol{\theta}_t) \quad // \text{ Update biased first moment estimate}$$
$$\mathbf{s}_{t+1} \leftarrow \beta_2 \cdot \mathbf{s}_t + (1 - \beta_2) \cdot \Delta J(\boldsymbol{\theta}_t) \odot \Delta J(\boldsymbol{\theta}_t) \quad // \text{ Update biased second moment estimate}$$

```


$$\tilde{\mathbf{m}}_{t+1} \leftarrow \frac{\mathbf{m}_{t+1}}{1-\beta}, \quad // \text{ Compute bias-corrected first moment estimate}$$

```

```


$$\tilde{\mathbf{S}}_{t+1} \leftarrow \frac{\mathbf{S}_{t+1}}{1 - \frac{1}{\rho}}$$

// Compute bias-corrected second moment estimate

```

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{s}}_{t+1} + \epsilon}} \odot \Delta J(\boldsymbol{\theta}_t) \quad // \text{ Update parameters}$$
until *convergence*

2.1.5 Stabilizing Gradients

The optimization of deep neural networks is prone to unstable gradients. It can either become noticeable in the form of vanishing or exploding gradients. This section discusses the most common approaches to alleviate these problems and to improve the training speed.

Batch Normalization

Batch normalization was introduced by Ioffe and Szegedy (2015) with the goal to accelerate the training of deep networks and reducing the internal covariate shift. Covariate shift refers to the change in the distribution of the input values to a learning algorithm. With batch normalization, an operation is added that normalizes the output of each layer before applying the activation function. Algorithm 5 shows that the mean μ_j and the variance σ_j^2 is calculated for each feature j . Each sample $x_{i,j}$ is mean-centered and divided by the standard deviation where ϵ is a small number for numerical stability in case the denominator becomes zero. The output $y_{i,j}$ of the batch normalization algorithm is then scaled by the parameter γ and shifted by the parameter β . Both parameters γ and β are learnable. The parameter β replaces the bias parameter in the previous layer as it would be a redundant shifting parameter.

Algorithm 5 Batch normalization

$$\mu_j \leftarrow \frac{1}{m} \sum_{i=1}^m x_{i,j}$$

$$\sigma_j^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_{i,j} - \mu_j)^2$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

$$y_{i,j} \leftarrow \gamma \cdot \hat{x}_{i,j} + \beta$$

Layer Normalization

Layer normalization is a method that has been developed in the work of Ba et al. (2016). It is highly related to batch normalization, however it does not normalize the input features across the batch dimension but across the features. Therefore the statistics are independent of other examples. For recurrent neural networks (see section 2.2) this is a great advantage because the statistics do not have to be computed for every time step. Additionally, this independence has the advantage over batch normalization that an arbitrary batch size can be used. At first glance, layer normalization stated in algorithm 6 looks similar to batch normalization. However, the statistics μ_i and σ_i^2 are calculated for each example i along the features axis j .

Algorithm 6 Layer normalization

$$\mu_i \leftarrow \frac{1}{m} \sum_{j=1}^m x_{i,j}$$

$$\sigma_i^2 \leftarrow \frac{1}{m} \sum_{j=1}^m (x_{i,j} - \mu_i)^2$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

$$y_{i,j} \leftarrow \gamma \cdot \hat{x}_{i,j} + \beta$$

Weights Normalization

Weight normalization was proposed by Salimans and Kingma (2016). Instead of normalizing the mini-batch they normalize the weights of the layers. To do so, the authors propose to parametrize each weight vector \mathbf{w} in terms of a parameter vector \mathbf{v} and a scalar parameter g according to equation 2.19 where $\|\mathbf{v}\|$ is the Euclidean norm of \mathbf{v} . They optimize both g and \mathbf{v} using gradient descent. The authors claim that this changes the learning dynamics and makes optimization easier. Moreover, the authors propose to combine weight normalization with a mean-only batch normalization. This variant of batch normalization only subtracts the mean of each mini-batch and does not divide it by the standard deviation.

$$\mathbf{w} = \frac{g}{\|\mathbf{v}\|} \mathbf{v} \quad (2.19)$$

Gradient Clipping

When applying the gradient descent algorithm, exploding gradients can occur. Goodfellow et al. (2016) describe gradient clipping as a technique to counter exploding gradients by clipping them in length. They limit the value of the gradients to a certain threshold value, which is a hyperparameter that has to be tuned.

Residual Blocks

The concepts of residual connections was first introduced by He et al. (2015) in the context of image recognition with the goal to simplify the training of very deep neural networks. Deep networks are likely to suffer from vanishing gradients. A residual block passes the activations from the previous layer to the next layer as a way of mitigating the vanishing gradient problem. Figure 2.10 illustrates the concept of residual connections.

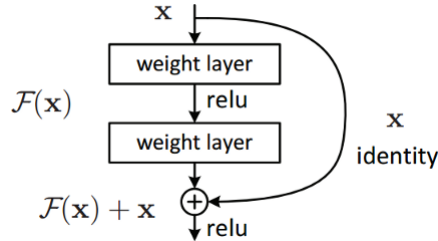


Figure 2.10: A visualization of a residual block by He et al. (2015)

2.2 Recurrent Neural Networks

Traditional neural networks accept a fixed-sized input vector and produce a fixed-sized output vector. To compute the output, they only rely on the current input, i.e. they don't use the results from the previous inputs to make its decisions. Elman (1990) explains that a recurrent neural network (RNN) is a neural network for sequences that addresses these issues. It computes an output sequence $\mathbf{y} = (y_1, \dots, y_T)$ using an input sequence $\mathbf{x} = (x_1, \dots, x_T)$ of variable length T and a hidden state \mathbf{h}_t . At each time step t , the hidden state \mathbf{h}_t is updated using the previous hidden state \mathbf{h}_{t-1} and the current input \mathbf{x}_t . Equation 2.20 shows how the hidden state is updated where the function f is a non-linear activation function or a complex RNN cell such as a gated recurrent unit (GRU) or a long short-term memory (LSTM) cell.

$$\mathbf{h}_t = \begin{cases} 0 & t = 0 \\ f(\mathbf{h}_{t-1}, \mathbf{x}_t) & \text{else} \end{cases} \quad (2.20)$$

In a standard recurrent cell the activation function f is usually a tanh function. Figure 2.11 shows a visualization of a simple RNN cell with the tanh activation function. The corresponding formula to compute the next hidden state \mathbf{h}_t is denoted in equation 2.21 where \mathbf{W}_x , \mathbf{W}_h and \mathbf{b}_h are learnable parameters.

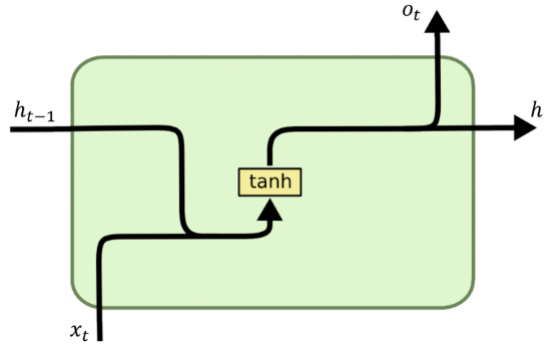


Figure 2.11: Visualization of a simple RNN cell by Colah (2015)

$$\mathbf{h}_t = \mathbf{o}_t = \tanh(\mathbf{W}_x \cdot \mathbf{x}_t + \mathbf{W}_h \cdot \mathbf{h}_{t-1} + \mathbf{b}_h) \quad (2.21)$$

An RNN is able to learn a probability distribution over a sequence by being trained to predict the next symbol in a sequence. The probability of a sequence \mathbf{x} can be represented as in equation 2.22. This learned distribution can be used to sample a new sequence by iteratively sampling a symbol at each time step t .

$$p(\mathbf{x}) = \prod_{t=1}^T p(\mathbf{x}_t | \mathbf{x}_{t-1}, \dots, \mathbf{x}_1) \quad (2.22)$$

RNNs are used whenever context from the previous input is needed. For simple RNNs this works well if only information from the recent past is required. However, this does not work well if a wider context is required.

Gated Recurrent Unit

To overcome the problem of long-term dependencies, more complex RNN cells were developed. The GRU cell was introduced by Cho et al. (2014) in the context of machine translation. A GRU cell consists of two gates: a reset gate \mathbf{r}_t and an update gate \mathbf{u}_t . The update gate controls what parts of the hidden state are updated or preserved. The reset gate controls what part of the previous hidden

state are used to compute the new content. Figure 2.12 shows a visualization of a GRU cell and algorithm 7 shows how to compute the next hidden state \mathbf{h}_t . The variable $\tilde{\mathbf{h}}_t$ denotes the candidate hidden state that is being weighted by the previous hidden state \mathbf{h}_{t-1} and the state \mathbf{z}_t .

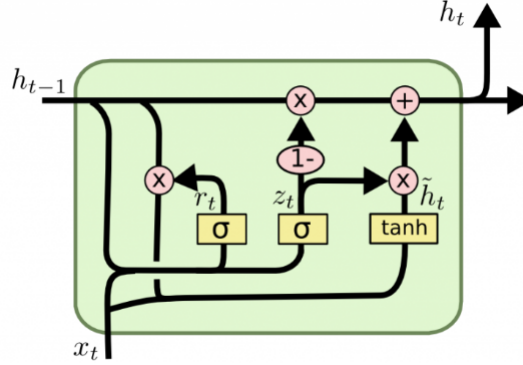


Figure 2.12: Visualization of a GRU cell by Colah (2015)

Algorithm 7 Compute the next hidden state \mathbf{h}_t for a GRU cell

$$\mathbf{z}_t \leftarrow \sigma(\mathbf{W}_z \cdot [\mathbf{h}_{t-1}; \mathbf{x}_t]) + \mathbf{b}_z$$

$$\mathbf{r}_t \leftarrow \sigma(\mathbf{W}_r \cdot [\mathbf{h}_{t-1}; \mathbf{x}_t]) + \mathbf{b}_r$$

$$\tilde{\mathbf{h}}_t \leftarrow \tanh(\mathbf{W} \cdot [\mathbf{r}_t \odot \mathbf{h}_{t-1}; \mathbf{x}_t]) + \mathbf{b}_h$$

$$\mathbf{h}_t \leftarrow (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t$$

Long Short-Term Memory

Long before the introduction of GRU cells, LSTM cells were proposed by Hochreiter and Schmidhuber (1997) to handle long sequences. In comparison to a GRU cell, an LSTM cell is more complex. It contains not only two, but three gates: a forget \mathbf{f}_t , an input \mathbf{i}_t and an output \mathbf{o}_t gate. The forget gate is used to control what information is forgotten and what is being kept. The input gate controls what parts of the new cell content are written. Lastly, the output gate is responsible for determining what parts of the cell are used as output to the hidden state. Figure 2.13 shows the internals of a LSTM cell. Algorithm 8 denotes the computation of the next hidden state \mathbf{h}_t in a LSTM cell where $\tilde{\mathbf{C}}_t$ is the candidate cell state at timestep t and \mathbf{C}_t the cell state at timestep t .

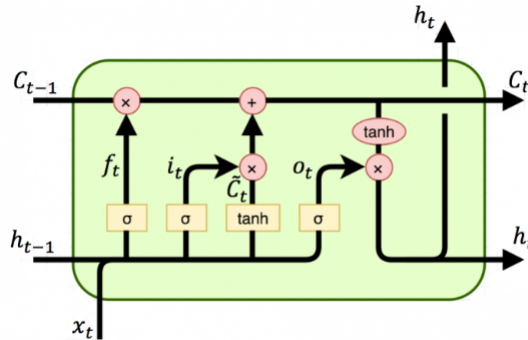


Figure 2.13: Visualization of a LSTM cell by Colah (2015)

Algorithm 8 Compute the next hidden state \mathbf{h}_t for a LSTM cell

$$\mathbf{f}_t \leftarrow \sigma(\mathbf{W}_f \cdot [\mathbf{C}_{t-1}; \mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_f)$$

$$\mathbf{i}_t \leftarrow \sigma(\mathbf{W}_i \cdot [\mathbf{C}_{t-1}; \mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_i)$$

$$\mathbf{o}_t \leftarrow \sigma(\mathbf{W}_o \cdot [\mathbf{C}_t; \mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_o)$$

$$\tilde{\mathbf{C}}_t \leftarrow \tanh(\mathbf{W}_c \cdot [\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_c)$$

$$\mathbf{C}_t \leftarrow \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t$$

$$\mathbf{h}_t \leftarrow \mathbf{o}_t \odot \tanh(\mathbf{C}_t)$$

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a special type of neural networks that contain convolution layers. They were proposed by Le Cun et al. (1989) for hand written digit recognition. Although CNNs were introduced in the late 1980s, they remained unpopular until Krizhevsky et al. (2012) used them to win the ImageNet challenge. The ImageNet challenge is an image classification challenge with the goal to classify images with more than 1'000 different classes. A CNN consists of several layers that are stacked onto each other. The following sections describe the most common layers.

2.3.1 Convolutional Layers

The main part of CNNs are convolutional layers, which have the property that they preserve the spatial structure of the input by passing it through a set of filters. A filter is moved across the input and multiplied with the the current part of the input. The products are then aggregated and result in a single scalar for each output. How many pixel a filter moves at each step is defined by means of the *stride size*. It is likely to occur that the configuration of the filter size and stride does not match, i.e. it is not possible to move the filter towards each input pixel. Therefore, there is the parameter *padding size*, that specifies the size of a zeroed frame added around the input.

Equation 2.23 shows how the output dimensions O_W and O_H can be calculated if a convolution layer with a filter size of $(w \times h)$, a padding size of P and a stride of S is applied to an input with the dimension $(W \times H)$.

$$O_W = \frac{W - w + 2 \cdot P_W}{S_w} + 1$$

$$O_H = \frac{H - h + 2 \cdot P_H}{S_h} + 1$$
(2.23)

Figure 2.14 demonstrates an example where a convolutional layer with filter size (3×3) is applied to an input with the dimension (5×5) . A zero padding of size 1 is applied and the filter is moved with a stride size of 2. This results in an output of the dimensions (3×3) .

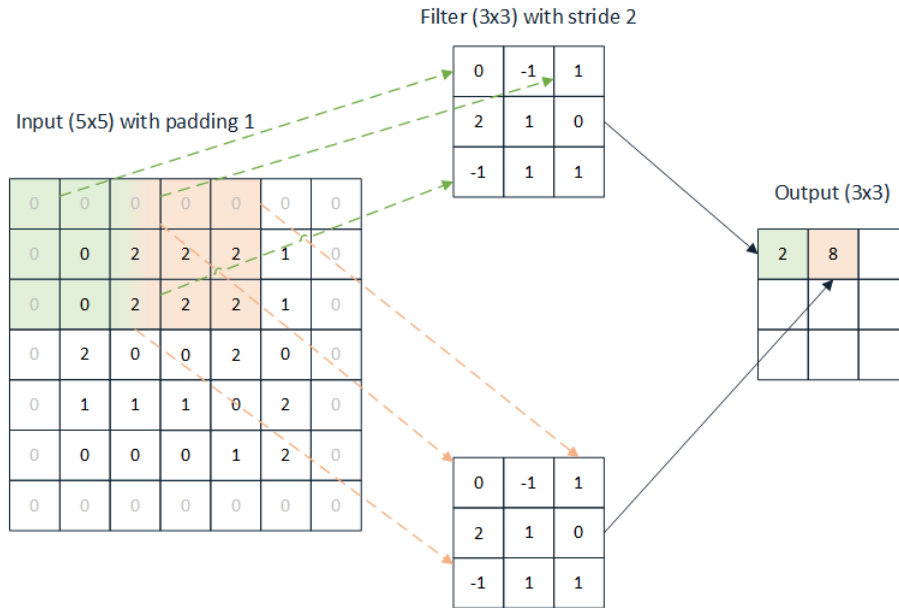


Figure 2.14: A CNN layer with size (3×3) , stride = 2 and padding = 1 applied to an input of dimension (5×5)

2.3.2 Pooling Layers

A pooling layer reduces the spatial size of the activation map. It is applied independently to every depth and is configured with the parameters *stride* and *pooling size*. The idea behind pooling layers is that most significant activations are kept while reducing the amount of computation.

Figure 2.15 shows an example of a max-pooling and an average-pooling with a pooling size of (2×2) and a stride size of 2. The max-pooling keeps the maximum value at each step whereas the average-pooling computes the average.

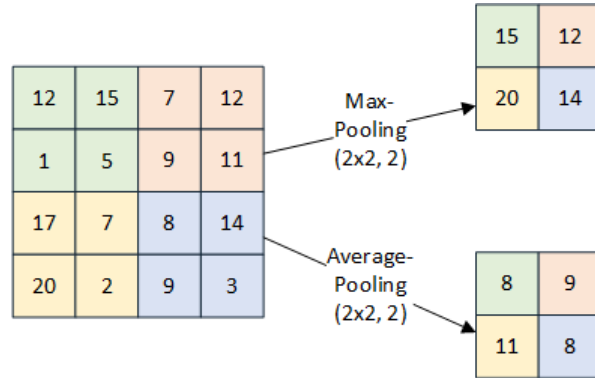


Figure 2.15: An example of max-pooling and average-pooling with a pooling size (2×2) and stride 2

2.4 Word Embeddings

Most of the algorithms for NLP systems cannot process text in its raw form. Instead, words must often be converted into numbers. This can be done by defining a vocabulary and then assigning a number for each word. Word embeddings are a projection of words into vectors of real numbers, i.e. a mapping of a space where each word has its own dimension to a space that is of lower dimensionality. A simple approach to create an embedding vector for each word is to one-hot encode every word. As a result, each word will have its own dimension. However, this results in a vector of the size of the vocabulary and is very inefficient because most of the values will be zero. Another disadvantage is that there is no notion of similarity between words, it is assumed that there is no relationship between them. This section presents several approaches to obtain word embeddings.

2.4.1 Term Frequency-Inverse Document Frequency

A more sophisticated approach to generate word embeddings is by counting the frequencies of each word. However, Jurafsky and Martin (2019) describe that raw word frequencies are very skewed and not discriminative. Therefore, using them as word embeddings is not very useful.

The term frequency-inverse document frequency (TF-IDF) weighting is a similar approach. Instead of simply counting the occurrences of each word in a single text, they are also counted in relation to the entire corpus. This allows for interpretation in how important a single word is to a corpus. Equation 2.24 shows how the TF-IDF for each word w is computed given a document d . It consists of two parts: the term frequency (TF) and the document frequency (DF). The function TF counts the occurrences of the word w in document d whereas DF counts the number of documents that contain w . The inverse document frequency (IDF) is used to give a higher weight to words that occur only in a few documents. In order to calculate the IDF, the number of documents N is divided by the DF.

$$\text{TF-IDF}(w, d) = \text{TF}(w, d) \cdot \log \left(\frac{N}{\text{DF}(w)} \right) \quad (2.24)$$

2.4.2 Word2Vec

Mikolov et al. (2013) proposed Word2Vec, a technique that allows to train word embeddings with a neural network. The main idea is that words with similar context should be located in close spatial dimensions. The author introduced two different models: the continuous bag of words (CBOW) and the skip-gram model. With the CBOW model, a neural network is trained to predict the most likely word given its context. The neural network is a shallow fully-connected network with only one hidden layer as depicted in figure 2.16. The input of the network is a one-hot encoded context word of size V . The hidden layer consists of N neurons and the output is the target word of size V . In the process of predicting the target words, a vector representation of each target word is being learnt.

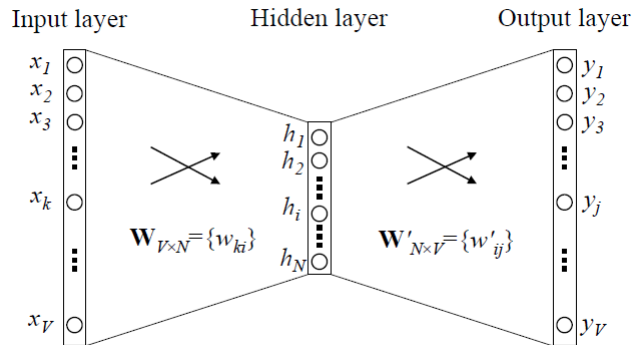


Figure 2.16: Visualization of the CBOW model by Rong (2014)

The idea of the skip-gram model is very similar to the CBOW, however, the neural network is trained to predict the context instead of the target word. According to the authors, the skip-gram approach works well with small amount of data and represents rare words well. Nevertheless CBOW is faster and the representation of frequent words is better.

After the network has been trained, a word embedding for a word in the vocabulary can be obtained by feeding it into the network. The output of the hidden layer represents the word in the embedding space. Mikolov et al. (2013) showed that with these learnt embeddings they were able to perform simple algebraic operations. For example, to find a word that is similar to *Italy* in the same sense as *France* to *Paris*, the word can be found by computing the vector \mathbf{X} as follows:

$$\mathbf{X} = \text{vector}(\text{"Paris"}) - \text{vector}(\text{"France"}) + \text{vector}(\text{"Italy"}) \quad (2.25)$$

Given the vector \mathbf{X} the most similar word can be found by computing the cosine similarity to each word in the vector space. The word with the highest cosine similarity represents the most similar word. In this example, it is very likely to be the word *Rome* and thus the answer to this question.

2.4.3 GloVe

Pennington et al. (2014) proposed global vectors (GloVe), a method that focuses on word co-occurrences over the whole corpus instead of sampling context and target words as in Word2Vec. The main principle behind GloVe is that the co-occurrence ratios between two words in a context are strongly connected to meaning. To generate the word embeddings, they first calculate a word-word co-occurrence matrix X_{ij} that defines the number of times i appears in the context of j .

Instead of using neural networks, the embeddings are directly optimized in a way that the dot product of two word vectors equal the logarithm of the number of times the word occurs near each other. The GloVe model minimizes the squared cost function showed in equation 2.26 using gradient descent where V is the size of the vocabulary. The weighting function $f(x)$ returns a value between 0 and 1 depending on the value of X_{ij} . The authors showed that using $\alpha = 3/4$ gives the best results. The terms \mathbf{w} , $\tilde{\mathbf{w}}$ are words vectors and \mathbf{b} , $\tilde{\mathbf{b}}$ are bias terms that capture the fact that some words occur more often than others.

$$J = \sum_{i,j=1}^V f(X_{ij}) \cdot (w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log(X_{ij}))^2 \quad (2.26)$$

$$f(x) = \begin{cases} (x/x_{max})^\alpha & \text{if } x < x_{max} \\ 1 & \text{else} \end{cases}$$

Although GloVe and Word2Vec use completely different methods for optimization, the authors showed that their results are surprisingly similar.

2.4.4 FastText

Word2Vec and GloVe have the limitation of not generalizing to unknown words. Bojanowski et al. (2017) introduced FastText, which is a very similar approach to Word2Vec. However, it does not learn vectors for words directly but represents each words as an n -gram of characters. A skip-gram model is trained to learn the embeddings using the n -gram representation of each word. This method helps to capture the meaning of short words and allows to understand suffixes and prefixes. If a word was not used in the training process, it can be divided into n -grams to obtain an embedding vector.

2.4.5 Flair

The approaches described above have the drawback that polysemous words are embedded into a single word vector. Polysemous words can have different meanings depending on its context. For example the word *bank* can be either a financial institute or a sloping land. Akbik et al. (2018) introduced Flair embeddings, which they refer as *contextual string embeddings*. Words and context are modelled as sequences of characters, which allows the model to handle rare and misspelled words. The embeddings are trained by using a character level language model (LM). The model architecture consists of an RNN with one LSTM cell. The goal of the LM is to predict the next character given an input sequence of characters. So the model possesses a hidden state for each character in the sequence.

To extract the embedding of a word, a single word lookup such as with the Word2Vec or GloVe approach is no longer possible as it requires to capture the context of the word. Figure 2.17 shows an example of the extraction of the flair embedding for the word *Washington*. Both hidden state outputs from the forward LM (shown in red) and the backward LM (shown in blue) are concatenated into a single word embedding.

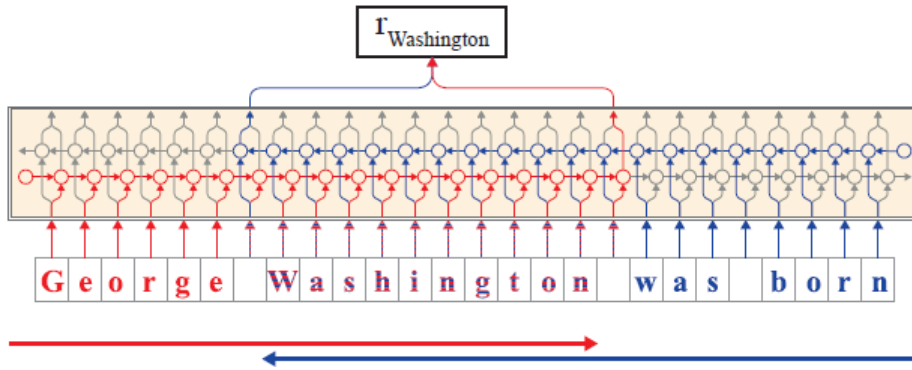


Figure 2.17: In illustration of the extraction of the flair embedding for the word *Washington* from the original flair paper by Akbik et al. (2018)

2.4.6 BERT

Bidirectional encoder representations from transformers (BERT) is a LM that is based on transformers (see section 2.6.4) and was introduced by Devlin et al. (2018). Like flair, it can be used to generate contextual word embeddings. Unlike standard LM that predicts the next word given the input sequence, BERT employs a masked LM task where 15% of the tokens are hidden. The model is then trained to predict the masked tokens. Thanks to this masked LM task, BERT has the unique property that it works bidirectional. To further improve the performance of the model, they developed the next sentence prediction (NSP) task. With the NSP task, the model has to predict the likelihood that sentence *B* belongs after sentence *A*.

2.5 Text Classification

Text classification is the process of assigning text documents to one or more categories. It is a fundamental task in the field of NLP and is closely related to sentiment analysis where the categories describe how the author feel about the topic of the text.

2.5.1 Rule-based Systems

With rule-based approaches, a text is being classified by using a set of handcrafted rules. The main advantage of rule-based systems is that they are comprehensible for humans. However, such systems require deep domain knowledge and are difficult to maintain. It therefore makes more sense to use approaches based on machine learning that learn from past observations.

2.5.2 Naïve Bayes Classifier

Pang et al. (2002) used the Naïve Bayes classifier in their work in the context of text classification. Naïve Bayes is a simple approach based on Bayes theorem. It assumes that the variables are independent as shown in equation 2.27.

$$P(x_1, x_2, \dots, x_N | c) = P(x_1 | c) P(x_2 | c) \dots p(x_N | c) = \prod_{k=1}^N p(x_k | c) \quad (2.27)$$

This leads to the formula for the Naïve Bayes classifier in equation 2.28 where C is the set of classes and N the number of features. In the context of text classification, the features are the occurrences of a certain word. Thus the prior probability $P(c)$ can be estimated as the number of words in a class divided by the total number of words as given in equation 2.29. The probability that a word occurs given a class c can be estimated by counting the number of occurrences of the word and dividing it by the total number of words in that class.

$$C_{NB} = \operatorname{argmax}_{c \in C} P(c) \prod_{k=1}^N P(x_k | c) \quad (2.28)$$

$$P(c) = \frac{\text{no. of tokens in } c}{\text{total no. of tokens}} \quad (2.29)$$

$$P(x_k | c) = \frac{\text{no. of occurrences of token } k \text{ in } c}{\text{total no. of tokens in } c}$$

2.5.3 Neural Approaches

Pang et al. (2002) showed that the Naïve Bayes Classifier performs surprisingly well, although its conditional independence does not hold in real-world situations. However, there are more sophisticated algorithms based on neural networks that achieve better results.

Recurrent Text Classification

Liu et al. (2016) describe the main steps of using RNNs to classify text as follows. The embedding is looked up for each input token and then passed into an RNN cell to obtain a fixed-sized vector. To deal with long-term dependencies, the RNN cells can be either LSTM or GRU layers. The output of the last hidden state of the RNN represents a fixed-sized vector that can be used by a fully-connected layer or an MLP to generate the output probabilities. The activation function of the last layer is typically a softmax for multi-class classification or a sigmoid for binary classification tasks.

Figure 2.18 shows an RNN that processes the input NOTAM "AIRPORT CLOSED DUE SNOW CLEARING". At each time step, the embedding vector of the input word is fed into the RNN to compute the next hidden state using the previous one. The last hidden state of the RNN represents the features of the NOTAM that can be used by a fully-connected layer to compute the output probability.

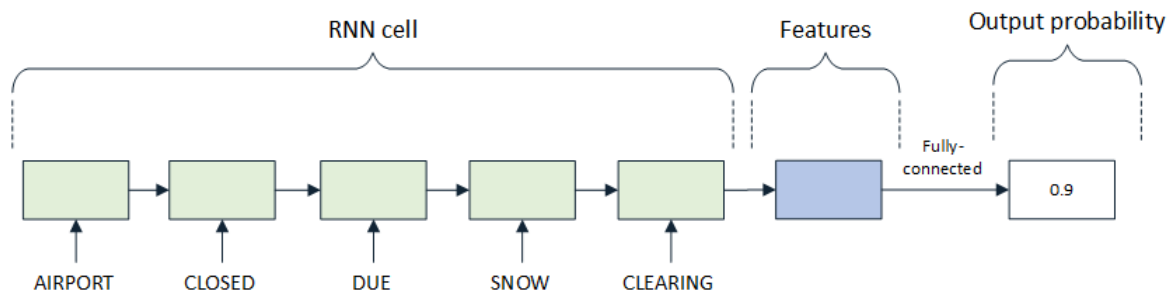


Figure 2.18: An example on how to extract features for text classification using an RNN

Convolutional Text Classification

The usage of CNNs has not only shown to achieve great results for computer vision tasks but also for a variety of NLP tasks, such as entity recognition or text classification. To classify text using CNNs, Jacovi et al. (2018) divides the required tasks into four steps:

1. Lookup the embeddings for the input.
2. Apply 1-dimensional convolutional filters that acts as n -gram detectors. Each filter is specialized for a family of n -grams.
3. Apply max-pooling over time to extract the relevant n -grams that are required to make a decision.
4. Use fully-connected layers to classify the text based on the extracted information.

The steps 2. and 3. can be done with multiple filter sizes in parallel. The resulting feature maps are then concatenated and fed into the fully-connected layers.

Figure 2.19 shows an example where the NOTAM is encoded into an embedding of dimension 3. Afterwards three 1-dimensional convolution filters of kernel size 2 are applied, resulting in a feature matrix of the dimensions (3×4) . Max-pooling over time returns the largest value for each filter which yields a (1×3) matrix that is fed into a fully-connected layer to compute the output probability.

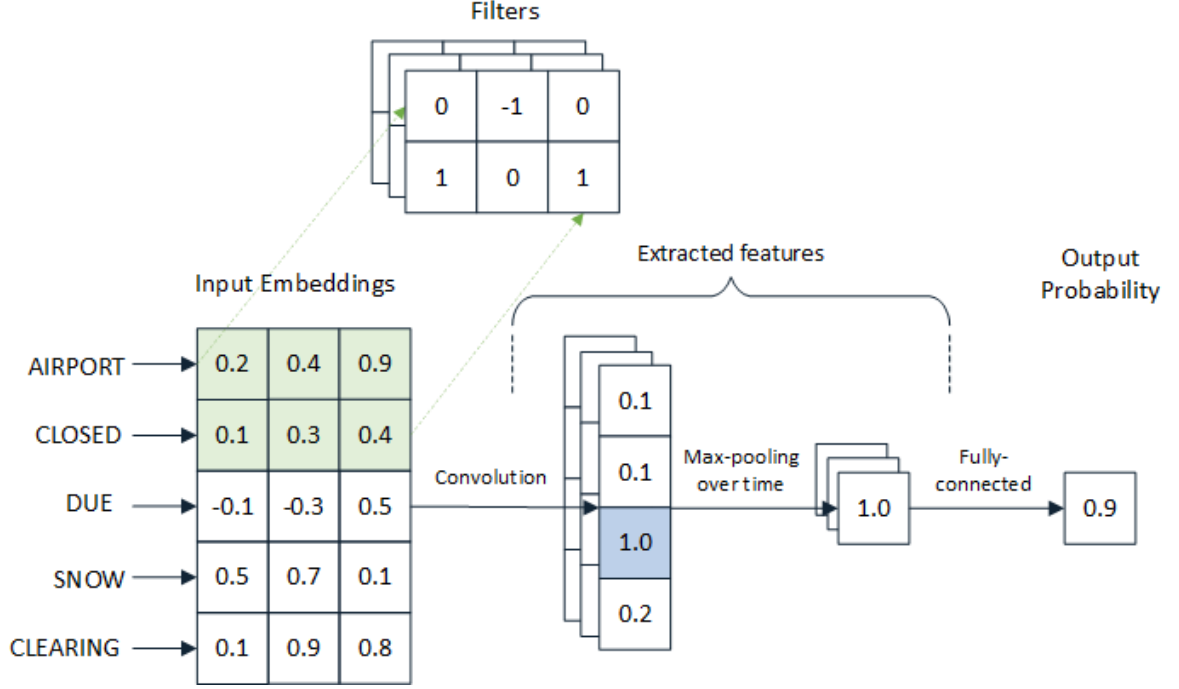


Figure 2.19: A text classification example using CNNs

Conneau et al. (2017) proposed the architecture VD-CNN, where they trained a very deep CNN for text classification. They do not operate on words, but directly on characters. Furthermore they use only small convolutions and pooling operations. In order to train such a deep CNN, they add residual connections after each convolutional block.

Combination of RNNs and CNNs for Text Classification

Wang et al. (2016) proposed a hybrid architecture consisting of RNNs and CNNs. They claim that with a CNN the model is able to extract local and deep features from the natural language which can be used by an RNN to learn long-distance dependencies. They first apply 1-dimensional convolutional filters with different filter sizes, followed by max-pooling, which acts as a feature selection procedure. The output of these convolutional layers is then concatenated and fed into an RNN to obtain a fixed-size vector. Lastly, the vector is used in a fully-connected layer to compute the output probabilities.

Figure 2.20 shows an example of such a hybrid model. The embeddings are obtained and then three 1-dimensional filters are applied which results in a feature matrix of the dimensions (3×4) . Max-pooling with pooling-size 2 and stride 2 is applied to get the most significant activations with dimension (3×2) . These are concatenated and fed into an RNN cell. The result of the RNN is used to calculate the output probability.

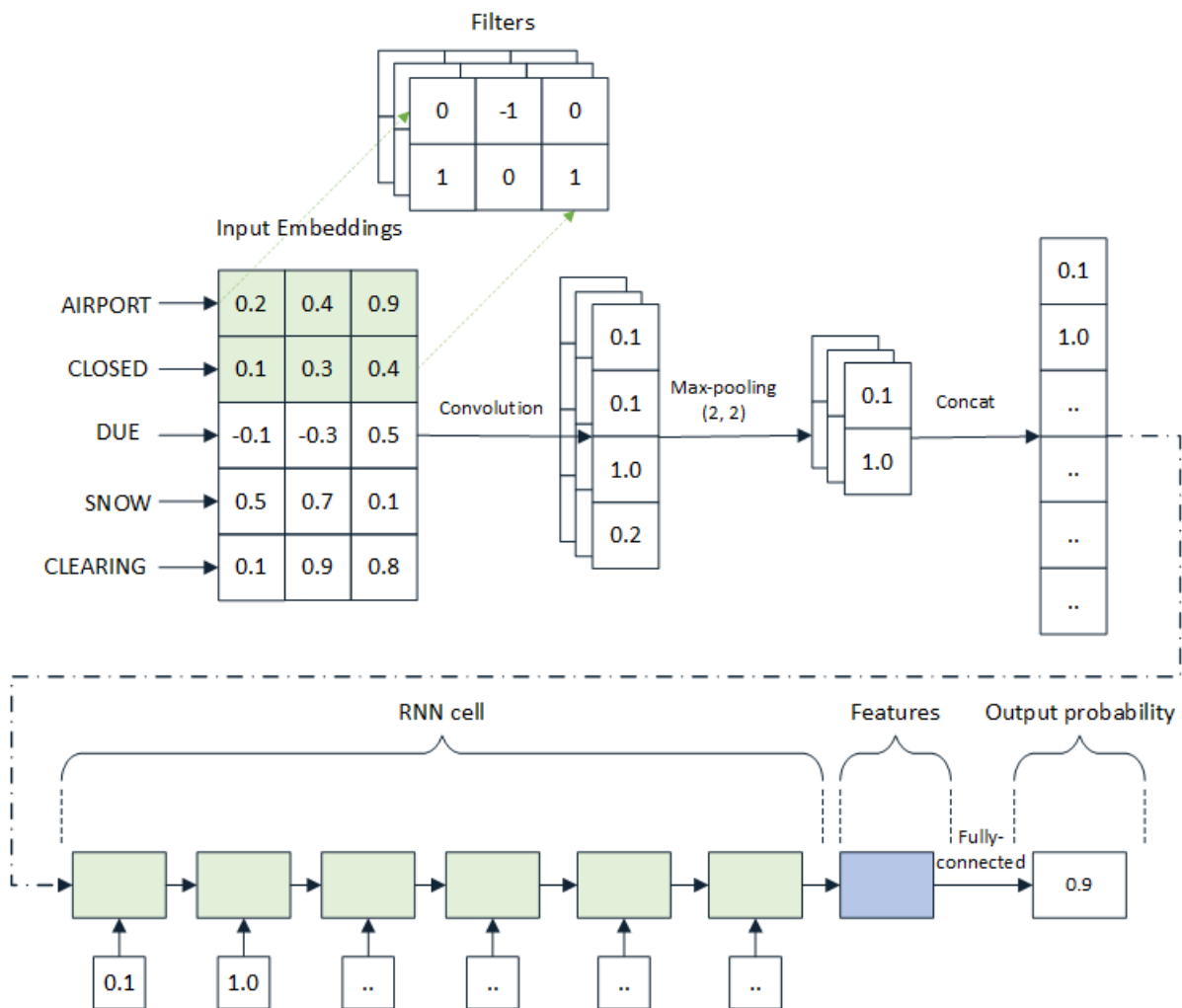


Figure 2.20: An example on how to extract features for text classification using CNNs

2.5.4 Metrics

To evaluate a text classification system, the conventional metrics for classification tasks can be used such as accuracy, precision, recall and F_β score. In order to compute these metrics, a confusion matrix has to be calculated first. Figure 2.21 shows how the confusion matrix is constructed.

		Predicted value	
		positive	negative
Actual value	positive	True Positive (TP)	False Negative (FN)
	negative	False Positive (FP)	True Negative (TN)

Figure 2.21: The confusion matrix

The most commonly used metric is the accuracy score. He and Garcia (2009) describe that the accuracy score has to be taken with care when the classes in the datasets are imbalanced. With an imbalanced dataset, a high accuracy is achieved if the classifier simply predicts the majority class for all examples. The recall score, or often called sensitivity or true positive rate (TPR), measures the completeness, i.e. how many examples of the positive class were predicted correctly. The precision score measures the exactness, i.e. how many of the examples predicted as positive were actually predicted correctly. The false positive rate (FPR) describes the probability of a false alarm and is the ratio between the false positives and the total number of actual negative events. The formulas are given below.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (2.30)$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (2.31)$$

$$\text{Recall} = \text{Sensitivity} = \text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (2.32)$$

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \quad (2.33)$$

The F_β score is the harmonic mean between precision and recall as denoted in equation 2.34. Common usage of the β parameter is setting it to 1, which defines the F_1 score. The F_1 score is the most frequently used metric for text classification tasks when using an imbalanced dataset.

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}} \quad (2.34)$$

In a multi-class classification setting, the metrics can be calculated in two manners: micro or macro average. With the micro average, the metrics are calculated globally by counting the total values of the confusion matrix. The macro average calculates the metric for each class and computes the mean value. It can be computed either weighted, depending on the number of true instances for each label, or unweighted.

2.6 Sequence to Sequence Models

The goal of sequence to sequence (Seq2Seq) models is to map an input sequence to an output sequence where the length of the sequences may differ. Seq2Seq models can be used for various tasks such as speech recognition, image captioning, text summarization or machine translation (MT).

MT is used to translate a source sentence written in a source language to a sentence written in a target language. From another perspective, MT can be viewed as finding a target sentence \mathbf{y} that maximizes the conditional probability of \mathbf{y} given a source sentence \mathbf{x} . With neural machine translation (NMT) a single neural network is trained to maximize the conditional probability of sentence pairs using a paired training corpus. A source sentence \mathbf{x} is translated by searching for the sentence that maximizes the conditional probability \mathbf{y} , i.e. $\text{argmax}_{\mathbf{y}} p(\mathbf{y}|\mathbf{x})$. Cho et al. (2014) proposed the first encoder-decoder NMT architecture where the source sequence is encoded into a fixed-sized vector and then decoded by a decoder. Sutskever et al. (2014) extended this approach and demonstrated that it can be used for other Seq2Seq tasks. The main problem of the previous approaches is that the performance decreases with the length of the sequences. Bahdanau et al. (2014) showed that by introducing an attention mechanism for the decoder, this problem could be alleviated. The main component of these architectures are RNNs, which face the problem that they cannot be easily parallelized. With the use of CNNs, authors like Kalchbrenner et al. (2016) showed that parallelization is possible. The state-of-the-art for Seq2Seq models are transformer networks proposed by Vaswani et al. (2017). The authors describe that their new architecture uses only the attention mechanism and simple feed forward neural networks. In the remainder of this section, different Seq2Seq models are explained and compared with each other.

2.6.1 Recurrent Sequence to Sequence Models

RNNs can be used to map sequences to sequences when the alignment between the input and the outputs is previously known. However, for MT this is not the case as a sentence in different languages might not be of the same size. Cho et al. (2014) suggested a Seq2Seq model based on two RNNs that act as an encoder and a decoder. They are jointly trained to maximize the conditional probability of the target sequence \mathbf{y} given an input sequence \mathbf{x} as stated in equation 2.35 where $y_{T'}$ is the last token of the target and x_T is the last token of the input sequence.

$$p(y_1, \dots, y_{T'} | x_1, \dots, x_T) \quad (2.35)$$

The encoder RNN f_e maps a variable-length input sequence \mathbf{x} to a fixed-sized vector \mathbf{c} . To do so, the encoder sequentially reads the input \mathbf{x} and updates the hidden state \mathbf{h}_t at each time step t according to equation 2.36. Using the hidden states, a fixed-sized encoder representation \mathbf{c} is obtained.

$$\begin{aligned} \mathbf{h}_t &= f_e(\mathbf{x}_t, \mathbf{h}_{t-1}) \\ \mathbf{c} &= q(\{\mathbf{h}_1, \dots, \mathbf{h}_{T_x}\}) \end{aligned} \quad (2.36)$$

Cho et al. (2014) use a GRU cell as the function f_e and the last hidden state as the summary vector \mathbf{c} , i.e. $\mathbf{c} = q(\{\mathbf{h}_1, \dots, \mathbf{h}_{T_x}\}) = \mathbf{h}_{T_x}$.

The decoder f_d is an additional RNN which is trained to generate the output sequence using the fixed-sized summary vector \mathbf{c} as input. The decoder network represents a language model for the target language. This is done by predicting the next symbol y_t given the hidden state \mathbf{s}_t of the decoder. To compute the hidden state \mathbf{s}_t , not only the previous hidden state \mathbf{s}_{t-1} but also the previous output y_{t-1} and the context vector \mathbf{c} are used as shown in equation 2.37.

$$\mathbf{s}_t = f_d(\mathbf{s}_{t-1}, y_{t-1}, \mathbf{c}) \quad (2.37)$$

The encoder and decoder networks are trained by maximizing the conditional log-likelihood (2.38) where θ represents the model parameters and (x_n, y_n) is a pair from the training set.

$$\max_{\theta} \frac{1}{N} \sum_{n=1}^N \log(p_{\theta}(y_n|x_n)) \quad (2.38)$$

The authors used the trained model for directly generating a target sequence and for scoring a given pair of input and output sequences in order to improve the performance of a traditional statistical machine translation (SMT) model.

Sutskever et al. (2014) proposed a universal end-to-end approach to map sequences to sequences similar to the one from Cho et al. (2014). The main difference is that they were using LSTM gates instead of GRU gates. Additionally, they discovered that the LSTM learns better when the source sentences are reversed. Rather than using the model for a traditional SMT model, they directly translate the source sentences with the encoder-decoder model.

Figure 2.22 shows an example of an encoder-decoder model where the model reads the input NOTAM "AIRPORT CLOSED DUE SNOW CLEARING" and produces "AP CLSD.". The encoder converts the input into a fixed-sized vector which is then fed into the decoder.

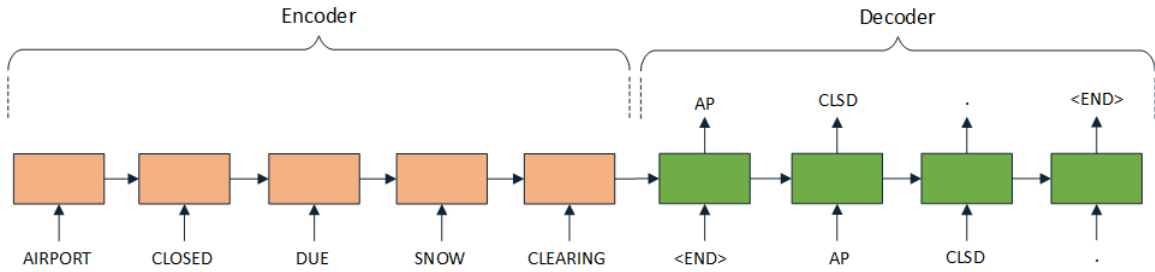


Figure 2.22: An encoder-decoder model

2.6.2 Recurrent Sequence to Sequence Models with Attention

The previously described methods encode the input sequence into a fixed-sized vector, which is usually represented by the last hidden state of a RNN cell. This is not ideal, since intermediate hidden states could contain useful information for the decoding task. It has been shown that the performance drops with increasing sequence length, i.e. the translation of long sentences does not work well. Bahdanau et al. (2014) proposed a novel architecture that addresses this bottleneck issue. The authors introduced an alignment model that assigns a score $\alpha_{i,j}$ to the pair of input at position i and output at position j based on how well they match. This allows the Seq2Seq model instead of encoding the entire sequence into a fixed-length vector, focus on the relevant information required to predict the next target word.

Figure 2.23 shows an example of an encoder-decoder model with attention where the model sequentially reads the input NOTAM and produces the output using an additional attention layer.

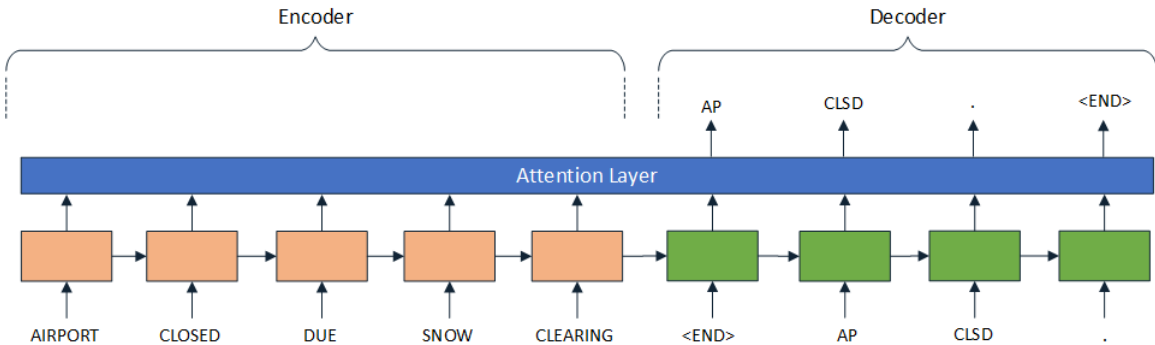


Figure 2.23: An encoder-decoder model with attention

In addition to the alignment model, the authors suggested to use a bidirectional RNN (biRNN) which consists of a forward and backward RNN. The forward RNN reads the sequence in its ordered way, the backward RNN reads the sequence in reverse order. The output of such a biRNN is a forward $\vec{\mathbf{h}}_j$ and a backward hidden state $\overleftarrow{\mathbf{h}}_j$ which the authors concatenate in a so-called annotation \mathbf{h}_j .

In order to generate the output sequence, the decoder uses the alignment model. At each step, the previous hidden state of the decoder \mathbf{s}_{i-1} and all the input annotations \mathbf{h}_j are fed into the alignment model. Bahdanau et al. (2014) parametrize the alignment model as a feed forward neural network which is often referred as additive attention. Luong et al. (2015a) introduced the multiplicative attention where the authors adapted the attention score function. Equation 2.39 shows how the attention scores are calculated, where both \mathbf{V}_a and \mathbf{W}_a are weight matrices that are jointly trained with all other components in the system.

$$e_{i,j} = \text{score}(\mathbf{s}_{i-1}, \mathbf{h}_j) = \begin{cases} \mathbf{V}_a^\top \tanh(\mathbf{W}_a[\mathbf{s}_{i-1}; \mathbf{h}_j]) & \text{Additive Attention} \\ \mathbf{s}_{i-1}^\top \mathbf{W}_a \mathbf{h}_j & \text{Multiplicative Attention} \end{cases} \quad (2.39)$$

The output $e_{i,j}$ of the alignment model is then fed into a softmax activation function to compute the attention weights α as denoted in equation 2.40. These weights represent the importance of the annotation \mathbf{h}_j and are used to calculate the context vector \mathbf{c}_i as shown in equation 2.41.

$$\alpha_{i,j} = \frac{\exp(e_{i,j})}{\sum_{k=1}^{T_x} \exp(e_{i,k})} \quad (2.40)$$

$$\mathbf{c}_i = \sum_{j=1}^{T_x} \alpha_{i,j} \cdot \mathbf{h}_j \quad (2.41)$$

This implements the attention mechanism where the decoder decides which parts of the source sentence it has to pay attention to. The information can now be spread throughout the annotations where the decoder filters for the relevant information. The attention matrix α can be plotted to explicitly show the correlation between the source and target words. Figure 2.24 illustrates an example of an alignment matrix where the model is tasked to convert the input "AIRPORT CLOSED DUE SNOW CLEARING." to "AP CLSD." The example shows that the model needs to pay attention to the word AIRPORT in order to generate the first token AP. Furthermore, it shows that the model generates the period symbol (.) by paying attention to the word DUE. This shows that the model learnt that it can remove the reason of the message.

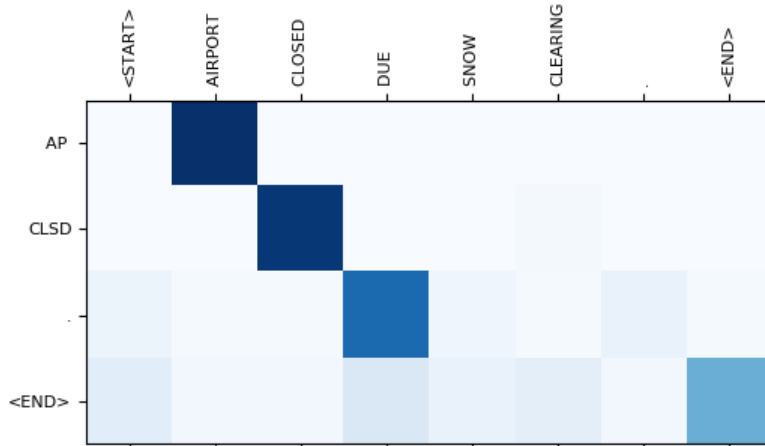


Figure 2.24: An example of an alignment matrix

Based on the context vector \mathbf{c}_t and the decoder hidden state \mathbf{s}_t of timestep t , a probability distribution is calculated over the fixed-sized vocabulary. Finally, the context vector and the decoder hidden states are concatenated and fed into a linear layer. A softmax activation function is applied to the result to obtain a probability distribution over the vocabulary words. Equation 2.42 shows how the final distribution over the vocabulary is obtained for each step where \mathbf{W}_v and \mathbf{b}_v are learnable parameters. Alternatively, the linear layer can be replaced by a MLP.

$$\mathbf{p}_{\text{vocab}} = \text{softmax}(\mathbf{W}_v[\mathbf{s}_t; \mathbf{c}_t] + \mathbf{b}_v) \quad (2.42)$$

Figure 2.25 shows how the next token "CLSD" is generated by using the attention mechanism. The context vector \mathbf{c}_t is obtained by the attention mechanism and is fed together with the current state of the decoder \mathbf{s}_t into a MLP to generate the vocabulary distribution.

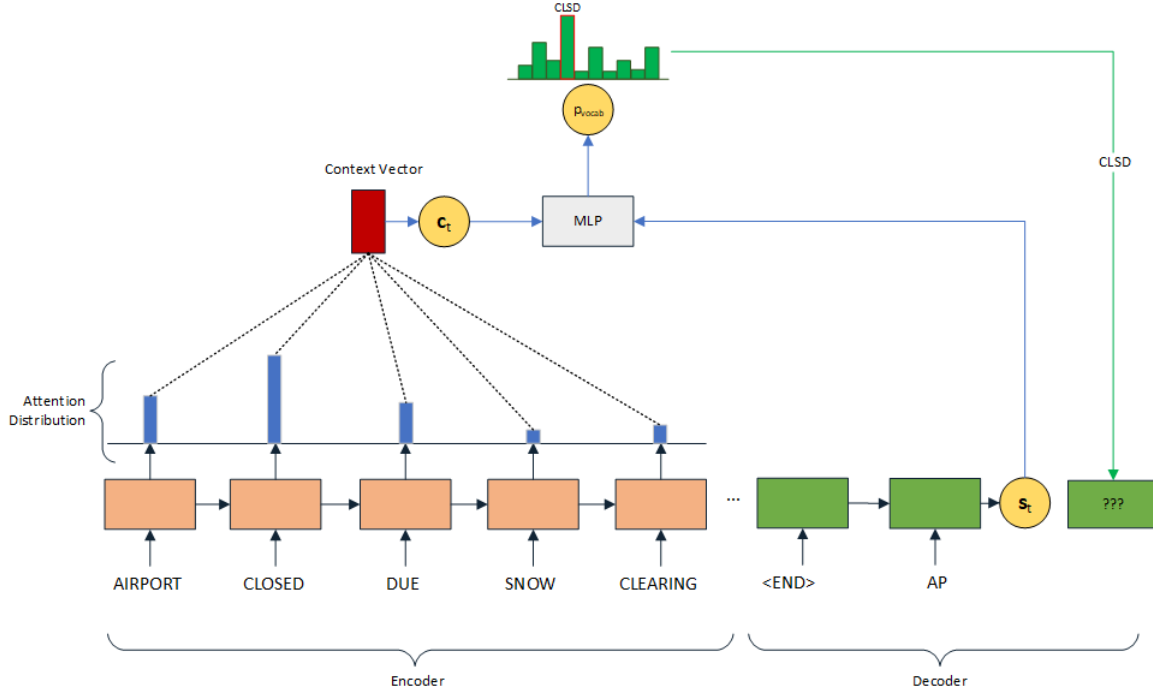


Figure 2.25: An example of a encoder-decoder network with attention to generate the next token

Maintaining Coverage for Seq2Seq Models

Tu et al. (2016) claim that the lack of coverage for Seq2Seq models is a serious problem. For the task of MT this could lead to wrong translations, because there is no mechanism to ensure that every single word is translated. This could result in either over-translation, where some words are unnecessarily translated more than once, or under-translation, where some words are skipped. For text summarization this is an even bigger problem because there could occur many repetitions for passages that are already summarized.

To alleviate these problems, the authors extended their attention-based Seq2Seq model to keep track of the attention history in form of a coverage vector \mathbf{C} . The calculation of the coverage vector is shown in equation 2.43 where f can be either a simple activation function such as \tanh or an RNN gating function. The authors used in their experiments a GRU cell.

$$C_{i,j} = f(C_{i-1,j}, \alpha_{i,j}, \mathbf{h}_j, \mathbf{s}_{i-1}) \quad (2.43)$$

See et al. (2017) adapt the calculation of the coverage vector by summing up the attention distribution over the previous time steps as shown in equation 2.44.

$$C_{i,j} = \sum_{i,j} \alpha_{i,j} \quad (2.44)$$

Using the coverage vector, the scoring function for additive attention can now be extended as follows:

$$e_{i,j} = \text{score}(\mathbf{s}_{i-1}, \mathbf{h}_j, C_{i-1,j}) = \mathbf{V}_a^\top \tanh(\mathbf{W}_a[\mathbf{s}_{i-1}; \mathbf{h}_j; C_{i-1,j}]) \quad (2.45)$$

To further discourage repetition, See et al. (2017) introduced a coverage loss to penalize repeatedly attending to the same locations. The coverage loss shown in equation 2.46 is weighted by some hyperparameter λ and added to the primary loss.

$$\text{covloss} = \sum_{i,j} \min(\alpha_{i,j}, C_{i,j}) \quad (2.46)$$

Pointer-Generator Networks

The use of Seq2Seq models for the task of text summarization has the main advantage that the network is able to summarize texts without simply rearranging passages from the original text. However, this approach has the disadvantage that factual details are often reproduced inaccurately. See et al. (2017) combined a pointer-network with an attention Seq2Seq model which they call pointer-generator network. The usage of a pointer-network allows the model to copy certain words directly from the source text whereas the classical Seq2Seq model (generator) allows the model to generate new content from the vocabulary.

During the decoding step, they estimate a probability \mathbf{p}_{gen} that indicates if the next word should be generated or copied from the source text by sampling from the attention distribution. Equation 2.47 denotes the computation of the generation probability \mathbf{p}_{gen} where \mathbf{c}_t is the context vector, \mathbf{s}_t the decoder state and \mathbf{x}_t the decoder input. The variables \mathbf{W}_c , \mathbf{W}_s and \mathbf{b}_{ptr} are learnable parameters that are trained with the model. The symbol σ denotes the sigmoid function that outputs a value between 0 and 1.

$$\mathbf{p}_{\text{gen}} = \sigma(\mathbf{W}_c^\top \mathbf{c}_t + \mathbf{W}_s^\top \mathbf{s}_t + \mathbf{W}_x^\top \mathbf{x}_t + \mathbf{b}_{\text{ptr}}) \quad (2.47)$$

To calculate the final output, they combine the generation probability \mathbf{p}_{gen} with the vocabulary distribution $\mathbf{p}_{\text{vocab}}$ as shown in equation 2.48. If the generation probability is one, the output probability is equal to the vocabulary distribution $\mathbf{p}_{\text{vocab}}$; if it is zero, the word is taken from the source sentence.

$$\mathbf{p}(w) = \mathbf{p}_{\text{gen}} \cdot \mathbf{p}_{\text{vocab}}(w) + (1 - \mathbf{p}_{\text{gen}}) \sum_{i:w_i=w} \alpha_{i,j} \quad (2.48)$$

Figure 2.26 demonstrates an example where the next token is generated by using a pointer-generator network. The generation probability \mathbf{p}_{gen} and the vocabulary distribution $\mathbf{p}_{\text{vocab}}$ are computed and then combined in a final distribution $\mathbf{p}(w)$. The item with the highest probability is the token "CLSD" which is generated. In this example, the word is being generated by the generator rather than copied by the pointer.

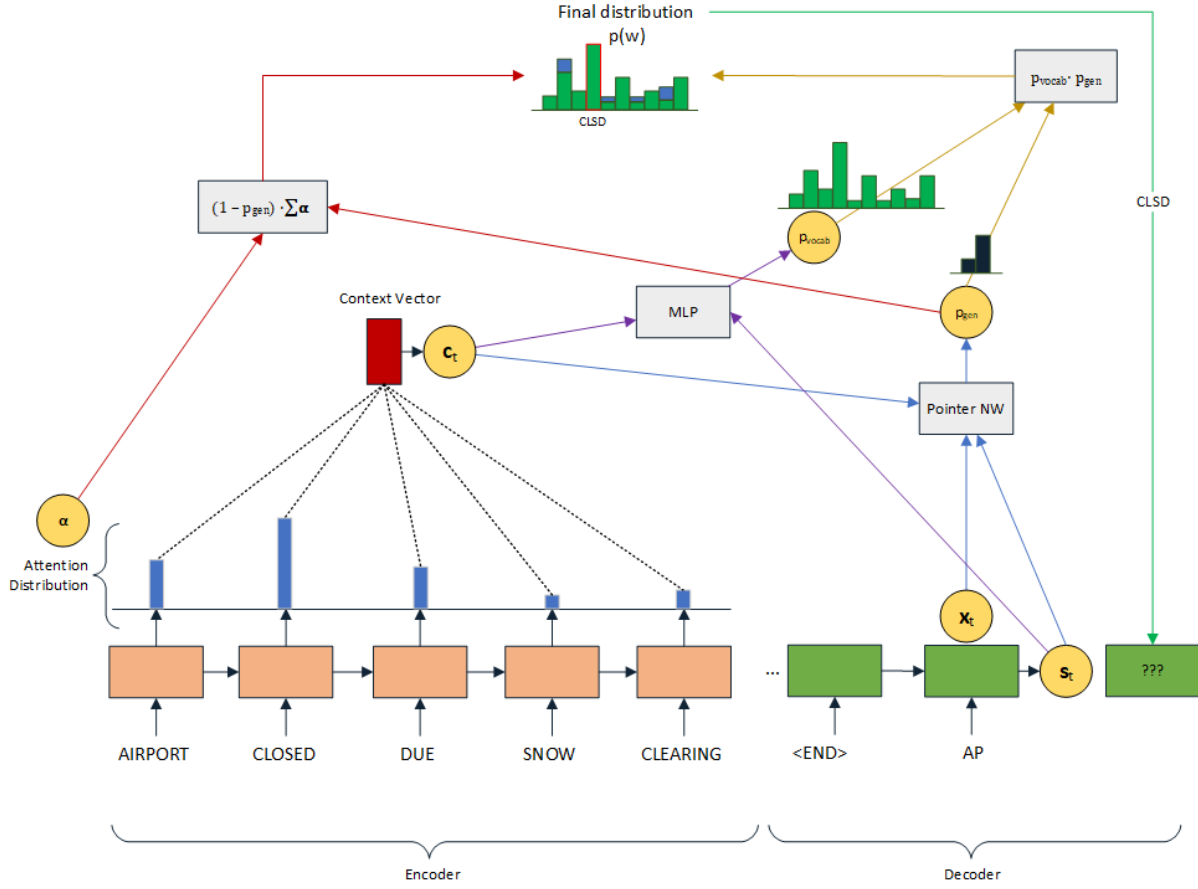


Figure 2.26: An example of a pointer-generator network

2.6.3 Convolutional Sequence to Sequence Models

The introduction of attention weights for RNN encoder-decoder models replaced the fixed-sized context vector with attention weights that allow the model to focus only on relevant information. This technique had a major impact on the performance of particularly long translations. These models rely heavily on the use of RNNs which have the disadvantage, that they cannot be parallelized along the sequence length. This is due to the sequential nature as they generate a sequence of hidden states \mathbf{h}_t as a function of the previous hidden state \mathbf{h}_{t-1} and the input x_t for position t . CNNs do not depend on the computation of the previous time step and therefore allow parallelization of every element in a sequence. Nevertheless, the use of CNNs for Seq2Seq models has some disadvantages: CNNs can not handle sequences of dynamic sizes and do not remember the temporal order of a sequence by design.

Gehring et al. (2016) presented an architecture where they replaced the RNN encoder with a CNN, which allows the encoding of the source sequence in parallel. However, their result could not beat the traditional RNN encoder-decoder models in terms of accuracy.

The ByteNet architecture, built as a one-dimensional CNN without the use of attention mechanism, was proposed by Kalchbrenner et al. (2016). The authors were able to perform NMT in linear time with respect to the length of the sequences. The source sequence is not encoded into a vector of constant size but is linear in the length of the source sequence. This is done by stacking the encoder and decoder networks, which has the effect of preserving the temporal resolution of the sequences. To handle sequences of different lengths, the authors implemented the *dynamic folding* mechanism where they defined a tighter upper bound for the target sequence by using a linear function. With the help of CNNs multiple words can be processed at the same time. The drawback is that the prediction of the current word can be affected from future words. The authors have shown that this issue can be resolved by masking the one-dimensional convolution in the decoder.

Gehring et al. (2017) extended their proposed architecture and replaced the RNN decoder with a CNN as well. They introduced the architecture called ConvS2S based entirely on CNNs that do not depend on the computation of the previous time step. In comparison to ByteNet, they made use of the attention mechanism. To give the model a sense of which portion of the sequence in the input or output is currently dealt with, the input sequence is positionally embedded. This is done by first obtaining the word embeddings from the input sequence and then by adding the absolute position of the input elements. The encoder and decoder are built with simple convolutional layers to compute intermediate states based on a fixed number of input elements. They showed that the number of operations to relate signals from two input or output positions grows logarithmically in the distance between positions, whereas in the ByteNet architecture the number of operations grows linearly.

2.6.4 Transformer Sequence to Sequence Models

With the use of CNNs for Seq2Seq models it is possible to encode and decode sequences in parallel. However, these models require a huge number of operations to relate signals from two arbitrary input or output positions. Vaswani et al. (2017) proposed a novel architecture that relies entirely on the attention mechanism instead of using RNNs or CNNs. The transformer architecture enabled the authors to reduce the amount of operations required to a constant number. The network contains a stack of $N = 6$ encoder and decoders. Figure 2.27 provides an overview of the proposed transformer network.

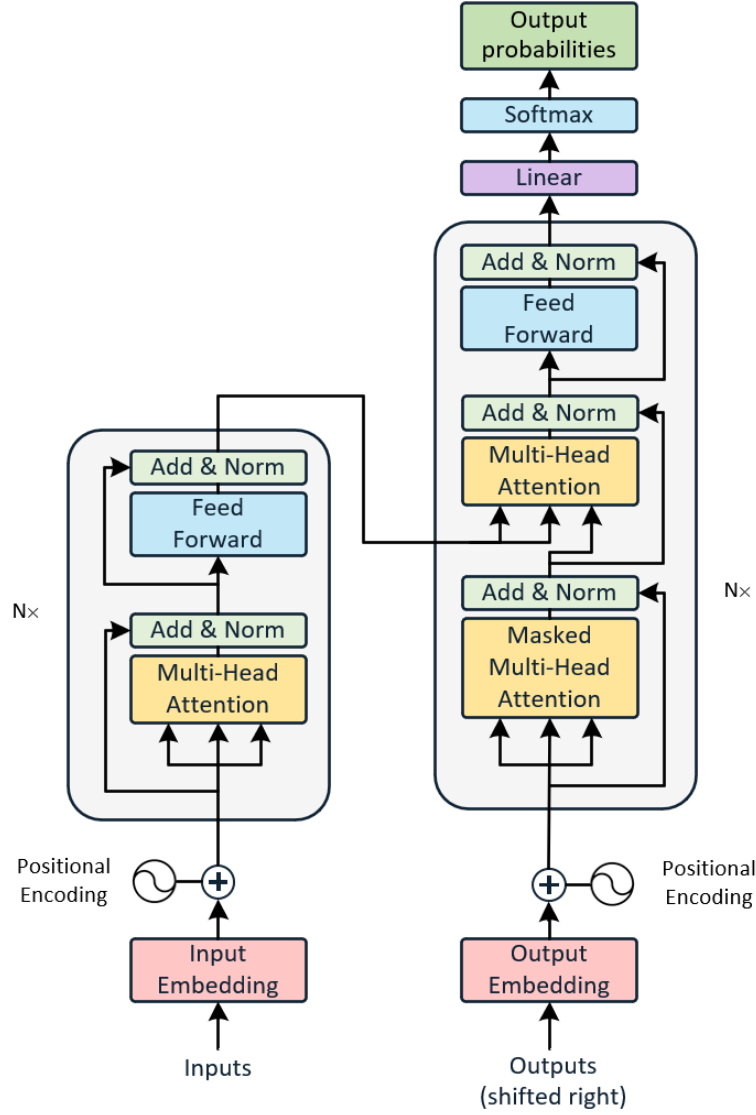


Figure 2.27: The architecture of the original transformer network proposed by Vaswani et al. (2017)

Encoder

The encoder consists of two main parts, self-attention and a fully-connected neural network. In order to reduce the training time and to stabilize the hidden states, layer normalization is performed after each layer. To account for the lack of recurrence in the transformer network, the authors proposed to positionally encode the input embeddings. They do this by injecting information about the relative position of the words in the sequence. These encodings are sine and cosine functions of different frequencies and are added to the input embeddings. This has the effect that locations have similar position-encoding vectors. Equation 2.49 shows such an encoding, where pos is the position and i the dimension.

$$\begin{aligned} \text{PE}_{(\text{pos}, 2i)} &= \sin(\text{pos}/10000^{2i/d_{\text{model}}}) \\ \text{PE}_{(\text{pos}, 2i+1)} &= \cos(\text{pos}/10000^{2i/d_{\text{model}}}) \end{aligned} \quad (2.49)$$

These encoded input embeddings are then passed to a multi-head attention layer. Between each layer, there is a residual connection which is used to retain the positional encodings. Self-attention is an attention mechanism that relates different positions of a single sequence and computes a representation of the same sequence. It allows the model to look at other positions in the input to obtain a better encoding for the current word.

The self-attention function is composed by the three vectors query \mathbf{Q} , key \mathbf{K} and value \mathbf{V} , which are created for each word of the encoder's input. There is a weight matrix for each vector: \mathbf{W}^Q for the query, \mathbf{W}^K for the key and \mathbf{W}^V for the value vector. These are learnable parameters that are trained. The vectors \mathbf{Q} , \mathbf{K} and \mathbf{V} are created by a matrix multiplication between the corresponding weight matrices and the input word embeddings and have usually a dimensionality of 64. The intuition of how much attention the model should pay to other words is done by scoring each word. To calculate the score, a "Scaled Dot-Product Attention" is calculated as shown in equation 2.50. The dot product between the key and query vector could grow large in magnitude and therefore lead to small gradients. To stabilize the gradients, the result is divided by the square root of the dimensionality of the key vector d_k before passing it to the softmax function.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{d_k}} \right) \mathbf{V} \quad (2.50)$$

This single self-attention mechanism is repeated $h = 8$ times to allow the model to jointly attend to information from different representation subspaces. The resulting attention vectors are then concatenated and multiplied with an additional weight matrix \mathbf{W}^O as stated in equation 2.51 and visualized in figure 2.28. The result of the multi-head attention is normalized and passed to the feed forward neural network.

$$\begin{aligned} \text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) &= \text{Concat}(\text{head}_1, \dots, \text{head}_h) \cdot \mathbf{W}^O \\ \text{where head}_i &= \text{Attention}(\mathbf{Q} \cdot \mathbf{W}_i^Q, \mathbf{K} \cdot \mathbf{W}_i^K, \mathbf{V} \cdot \mathbf{W}_i^V) \end{aligned} \quad (2.51)$$

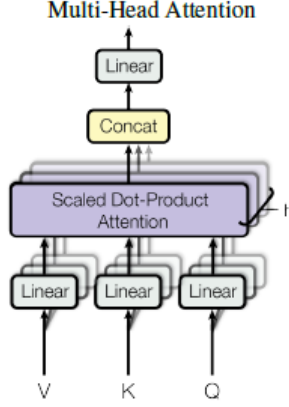


Figure 2.28: A visualization of the computation graph of the multi-head attention in Vaswani et al. (2017)

Decoder

The decoder is almost identical to the encoder but has an additional part, an encoder-decoder attention layer. This layer is similar to the Seq2Seq models with attention, which helps the decoder to focus on relevant parts of the input sequence. The query vector is created with the layer below and the key and value vectors correspond to the output of the encoder. By masking future positions, the attention layer can only attend to earlier positions in the output sequence. Finally, the output of the decoder is fed into a linear layer with a softmax activation function to transform it into a probability distribution of words with the size of the vocabulary $\mathbf{p}_{\text{vocab}}$.

Transformer Pointer-Generator Models

Deaton et al. (2018) extended the transformer model by introducing a pointer-network that allows the model to copy words from the input sequence. The output of the pointer-network are generation probabilities \mathbf{p}_{gen} which represent the probability of each token being generated. They used the technique from See et al. (2017), which has been described in section 2.6.2, to parametrize the pointer-generator network. The following equation is used to calculate the generation probabilities:

$$\mathbf{p}_{\text{gen}} = \sigma(\mathbf{W}_c^\top \mathbf{c}_t + \mathbf{W}_s^\top \mathbf{s}_t + \mathbf{W}_x^\top \mathbf{x}_t + \mathbf{b}_{ptr}) \quad (2.52)$$

The authors mapped \mathbf{s}_t , \mathbf{c}_t and \mathbf{x}_t to analogous values from the transformer model. Since the transformer model outputs attention distributions α for each head, the authors sum them up across the multiple heads. The value \mathbf{x}_t is the input of the decoder. In order to generate the context vector \mathbf{c}_t , they use the output of the final encoder layer, weight them according to the attention weights and take the average across the source dimension.

2.6.5 Handle Out-of-Vocabulary Words

The first step when training a Seq2Seq model for MT or text summarization is the conversion of the corpus into a numerical representation. This is often done by assigning a scalar to each word or use its embedding representation. However, during testing time there may be a word in the sentence that the model has never seen. These rare words are called out-of-vocabulary (OOV) words. Regardless of the chosen model architecture, developers have to implement a way of handling OOV words.

Unk Symbol The simplest way is to previously define a symbol that stands for all OOV words. During testing time, the model outputs this symbol for each word that was out-of-vocabulary.

Alignment Model Luong et al. (2015b) propose a more sophisticated way, where they train an alignment model parallel to the NMT system, which emits for each OOV word the position of the corresponding word in the source sentence. During testing time, they add a post-processing step where they identify each OOV word and its position. To translate the OOV words, they are queried in a back-off dictionary.

Large Vocabulary Trick A different approach to reduce the amount of OOV words would be to use a very large vocabulary. However, training as well as the decoding complexity increases proportionally to the number of target words. Jean et al. (2015) developed an importance sampling technique that allows to use only a small subset of the whole vocabulary to decode a sequence. This way they were able to use a very large target vocabulary without increasing training complexity.

Character Embeddings Another common approach is to train character embeddings instead of word embeddings. Lee et al. (2017) presented a fully character-level NMT model that maps a character sequence in a source language to a character sequence in a target language.

Subword Units An approach that lies between the word embeddings and the character embeddings has been proposed by Sennrich et al. (2015). They claim that various words are translatable via smaller units than words. They segment rare and unseen words into subwords and use them to train an NMT model. For example the word *subword* is split into the words *sub* and *word*. The authors used the byte pair encoding (BPE) algorithm to build the subword dictionary. The BPE allows to define a desired target vocabulary size. A major disadvantage of using a subword encoding is that training becomes more difficult the longer the sequences are.

Pointer Softmax Gulcehre et al. (2016) presented a way to deal with rare and unseen words by introducing a new layer called pointer softmax (PS). They do this by extending the output of their model with an additional softmax layer that predicts the location of a word in the source sentence. They use a MLP which acts as a switching network that outputs a binary variable. This binary variable indicates whether to use the softmax output that points to the location of the word in the input sequence or the translated word. They particularly use this method to ensure that words for which there is no translation, such as names, are copied completely from the source sequence. This method has been further extended by See et al. (2017) with the introduction of pointer-generator networks described in section 2.6.2.

2.6.6 Beam Search

Regardless of using RNNs, CNNs or transformer models, the aforementioned architectures have something in common: they all use a decoder that is autoregressive, i.e. generating a token is conditioned on the previously generated tokens. For each word of the target sequence, the decoder predicts a multinomial probability distribution over the vocabulary. The goal of the decoder is to maximize the joint probability of the target sequence \mathbf{y} , given the source sequence \mathbf{x} as denoted in equation 2.53.

$$P(\mathbf{y}|\mathbf{x}) = \prod_{t=1}^T P(y_t|y_1, \dots, y_{t-1}, \mathbf{x}) \quad (2.53)$$

To produce a translated sequence, the next word must be selected and then used to predict the following word. There are two search strategies to select the next word: greedy search and beam search. The greedy approach selects the most likely word given the predicted probability distribution. This is a rather intuitive way, but it only maximizes the probability for each next word in isolation. However, it can lead to bad translations, especially if the model makes a mistake in a single step.

The second approach is called beam search. Sutskever et al. (2014) uses a left-to-right beam search decoder that does not only select the word with the highest probability, but also keeps track of B partial hypotheses. At each timestep, every partial hypothesis in the beam is extended with all words in the vocabulary. Afterwards only the most likely B partial hypotheses are kept. This is repeated until the end-of-sequence token occurs for every hypothesis or a predefined maximal number of steps is reached. As soon as all hypotheses are completed, all but the hypothesis with the highest score are discarded. The joint probability of the whole sequence would be maximized by setting B equal to the number of words in the vocabulary; i.e. at each step, every single word is kept. However, this exhaustive search is not feasible as it would require an enormous amount of memory. When setting B equal to 1, this approach is similar to the greedy search. There is no universal beam size B that performs well for all use cases. Therefore, it is treated as a hyperparameter that must be tuned. To prevent numerical underflow, the score of each hypothesis is not calculated by multiplying the probabilities, but by summing the logarithm of the probabilities.

Figure 2.29 shows an example of the beam search algorithm with beam size $B = 2$. The tree is extended until the $\langle \text{END} \rangle$ token occurs for every hypothesis. The hypothesis " $\langle \text{START} \rangle$ AP CLSD . $\langle \text{END} \rangle$ " achieves the highest sum of log-probabilities and is therefore selected.

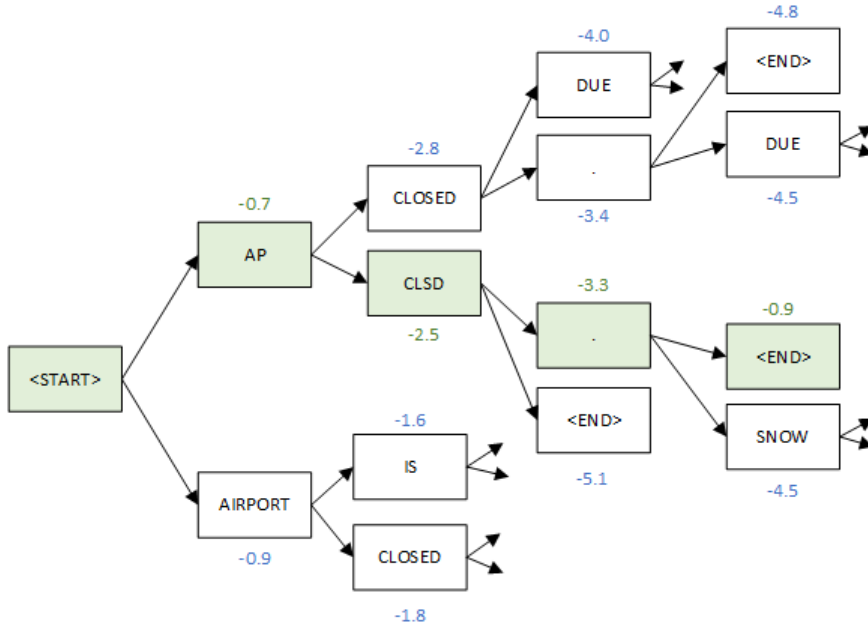


Figure 2.29: An example of a beam search decoder with beam size 2

Wu et al. (2016) refined the pure maximum probability beam search by introducing a length normalisation. The reason behind it is that beam search favours shorter translations over longer ones since adding a negative log-probability decreases the score of the hypothesis. Equation 2.54 shows how the score of each hypothesis is normalised by dividing it by a length penalty $\text{lp}(\mathbf{y})$. The authors suggested to use an α value between 0.6 and 0.7 for the length penalty.

$$\text{score}(\mathbf{y}, \mathbf{x}) = \frac{\log(P(\mathbf{y}|\mathbf{x}))}{\text{lp}(\mathbf{y})} \quad (2.54)$$

$$\text{lp}(\mathbf{y}) = |\mathbf{y}|^\alpha$$

Alternatively, they suggested to use a length penalty which they have developed empirically as follows:

$$\text{lp}(\mathbf{y}) = \frac{(5 + |\mathbf{y}|)^\alpha}{(5 + 1)^\alpha} \quad (2.55)$$

When using attention models, the authors proposed to add a coverage penalty factor $\text{cp}(\mathbf{x}, \mathbf{y})$ that penalizes outputs which do not pay enough attention to input tokens:

$$\text{score}(\mathbf{y}, \mathbf{x}) = \frac{\log(P(\mathbf{y}|\mathbf{x}))}{\text{lp}(\mathbf{y})} + \text{cp}(\mathbf{x}, \mathbf{y}) \quad (2.56)$$

Equation 2.57 shows how the penalty is calculated where in this case \mathbf{p} is the attention matrix and β a value between 0 and 1 in order to weight the coverage penalty.

$$\text{cp}(\mathbf{x}, \mathbf{y}) = \beta \cdot \sum_{i=1}^{|\mathbf{x}|} \log \left(\min \left(\sum_{j=1}^{|\mathbf{y}|} p_{i,j}, 1.0 \right) \right) \quad (2.57)$$

For the final scoring function, the authors achieved the best result with the parameters $\alpha = 0.2$ and $\beta = 0.2$ with their experiments.

To avoid repetitions in the generated output, Deaton et al. (2018) implemented the so-called n -gram blocking approach. With this approach, a hypothesis is discarded if it contains a certain n -gram more than once. They parametrized the n -gram blocking with $n = 2$.

2.6.7 Metrics

NLP classification tasks where the output is a label are easy to evaluate. By means of a simple label matching, metrics like accuracy, precision or recall can be calculated. Evaluating NLP tasks such as text summarization or MT is much more complex. To evaluate the performance of an MT system, humans have to compare the output of the systems with reference translations. This takes a lot of time and is often a problem for developers, as they need to check the performance of their systems regularly. Therefore, a number of different evaluation metrics have been proposed by several researchers, which are described in this section.

Word Error Rate

One of the most intuitive metrics is the word error rate (WER). It is based on the Levenshtein distance, a metric for measuring the difference between two sequences. Marzal and Vidal (1993) describe the Levenshtein distance as the minimum number of edits required to transform the output into the reference where an edit can be a substitution, a deletion or an insertion. The Levenshtein distance works on character level, whereas the WER on word level. Equation 2.58 shows how the WER is calculated where S is the number of substitutions, D the number of deletions and I the number of insertions. The term C represents the number of correct words and N is the number of words in the reference. The WER has the main drawback that it only considers one sentence to be correct and is computationally expensive.

$$\text{WER} = \frac{S + D + I}{N} = \frac{S + D + I}{S + D + C} \quad (2.58)$$

BLEU Score

The bilingual evaluation understudy (BLEU) score was introduced by Papineni et al. (2002). It is an MT evaluation metric that is quick, inexpensive and language-independent. The authors showed that it correlates highly with human evaluation. The central idea of their proposal is that *"the closer a machine translation is to a professional human translation, the better it is"*. They measure the closeness to one or more reference translations according to a numerical metric. The main challenge is that there are many "perfect" translations of a source sentence, which may have different words or word orders.

They first introduced a baseline BLEU score that computes the precision by counting the number of n -gram words which occur in any reference translation and then divide it by the total number of words in the translation. An n -gram is a contiguous sequence of n items from a given text. For example all words of *"The cat is on the mat"* are n -grams of size 1 (unigrams). Word tuples like *(The cat)*, *(cat is)* are n -grams of size 2 (bigrams). Table 2.1 shows an example where the unigram precision is $7/7 = 1$, because each of the seven words in the translation appear in the reference translation.

Reference translation	Translation
The cat is on the mat	the the the the the the the
There is a cat on the mat	
Unigram precision = $7/7$	Modified unigram precision $p_n = 2/7$

Table 2.1: BLEU score example for unigram precision

This is clearly not a good evaluation metric, thus the authors introduced the modified n -gram precision p_n . It can be calculated by first counting the maximum number of times a word occurs in any single reference translation. Next, the total count of each translated word is clipped by its maximum reference count. These clipped counts are added up and then divided by the total number of translated words. The modified unigram precision for the example in table 2.1 is now $2/7$ instead of $7/7$. The authors have shown p_n decays exponentially with n . Yet in all cases, they were able to distinguish between a good and a bad translation. So it makes sense to combine multiple p_n into a single number. Equation 2.59 shows how the modified n -gram precision is calculated for an arbitrary size of n where \hat{y} is the output of the MT system. The $\text{Count}_{\text{clip}}$ is defined as the minimum between the word count and largest count observed in any single reference for that word.

$$\begin{aligned} \text{Count}_{\text{clip}} &= \min(\text{Count}, \text{MaxRefCount}) \\ p_n &= \frac{\sum_{\text{gram}_n \in \hat{y}} \text{Count}_{\text{clip}}(\text{gram}_n)}{\sum_{\text{gram}'_n \in \hat{y}} \text{Count}(\text{gram}'_n)} \end{aligned} \quad (2.59)$$

Translations that are longer than their references are penalized by the modified n -gram precision. To penalize shorter translations, the authors have introduced a multiplicative brevity factor. As stated in equation 2.60, the brevity penalty factor (BP) is 1 if the length of the translation c is larger than the reference r and exponentially decaying in r/c otherwise.

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases} \quad (2.60)$$

The authors used multiple n -grams and averaged them with a geometric mean. Equation 2.61 shows that the BLEU score is calculated by multiplying the BP with the exponential sum of the logarithm of the modified n -gram precisions p_n weighted by w_n . Common parameters are $N = 4$ and uniform weights $w_n = 1/N$.

$$\text{BLEU}_N = \text{BP} \cdot \exp \left(\sum_{n=1}^N w_n \log(p_n) \right) \quad (2.61)$$

The original BLEU score that the authors proposed is a corpus score. To calculate the BLEU score of a MT system it is not intended to calculate the score for each sentence and then taking an average. Instead the numerators and denominators for each translation-reference pair are summed up before the division, i.e. the BLEU score calculates the micro-average precision.

There are many adaptations of the BLEU score, such as the NIST score suggested by Doddington (2002). The NIST score further calculates how informative a particular n -gram is by counting how many times a particular n -gram occurs. A more recent modification is the google BLEU (GLEU) score which has been used in Wu et al. (2016). The GLEU score should solve the problem that the BLEU score is corpus-based. For each n -gram they calculate not only the precision but also the recall as the number of matching n -grams divided by the number of total n -grams in the reference. The GLEU score is then the minimum between the precision and recall.

ROUGE Score

The recall-oriented understudy for gisting evaluation (ROUGE) package has been suggested by Lin (2004) and is a collection of evaluation metrics for text summarization. Four different metrics exist: ROUGE-N, ROUGE-L, ROUGE-W and ROUGE-S. In the following section the ROUGE-N score is described and therefore referred to ROUGE score.

The ROUGE score can be seen as a modified BLEU score that focuses on recall rather than on precision. Instead of counting how many of the n -grams in the output appear in the references, the ROUGE score counts how many n -grams of the reference appear in the output.

$$\text{ROUGE}_n \text{ Recall} = \frac{\sum_{S \in \{\text{ReferenceSummaries}\}} \sum_{\text{gram}_n \in S} \text{Count}_{\text{match}}(\text{gram}_n)}{\sum_{S \in \{\text{ReferenceSummaries}\}} \sum_{\text{gram}_n \in S} \text{Count}(\text{gram}_n)} \quad (2.62)$$

Equation 2.62 states how the ROUGE_n Recall can be calculated where n is the size of the n -grams. The function $\text{Count}_{\text{match}}$ returns the number of n -grams that occur both in the reference and in the output. This sum is divided by the total sum of the number of n -grams occurring in the reference. If the output of the Seq2Seq model is really long, the ROUGE score would be large, as it only captures the recall. Therefore, several authors such as Ganesan (2018) additionally calculate the ROUGE_n

precision score, which can be calculated by dividing the number of overlapping n -grams by the total n -grams in the output as derived in equation 2.63.

$$\text{ROUGE}_n \text{ Precision} = \frac{\sum_{S \in \{\text{ReferenceSummaries}\}} \sum_{\text{gram}_n \in S} \text{Count}_{\text{match}}(\text{gram}_n)}{\sum_{\text{gram}_n \in \text{OutputSummary}} \text{Count}(\text{gram}_n)} \quad (2.63)$$

With the precision and recall at hand, the $\text{ROUGE}_n F_1$ score is calculated as the harmonic mean between the recall and precision as shown in equation 2.64.

$$\text{ROUGE}_n F_1 = 2 \cdot \frac{\text{ROUGE}_n \text{ Precision} \cdot \text{ROUGE}_n \text{ Recall}}{\text{ROUGE}_n \text{ Precision} + \text{ROUGE}_n \text{ Recall}} \quad (2.64)$$

METEOR Score

Both aforementioned metrics BLEU and ROUGE do not take the ordering of words into account. Banerjee and Lavie (2005) claim that the lack of recall for the BLEU score cannot be simply compensated by means of a static BP factor. Therefore they proposed the metric for evaluation of translation with explicit ordering (METEOR) score and allege that it has better correlation with human judgement.

The METEOR score is computed by explicit unigram matches between the translation and the reference. Those alignments are done in three stages: actual matching, a porter-stem stage and a synonym stage. In the first stage, only exact matches are considered. In the porter-stem stage, the words are stemmed with a porter-stemmer before matching. Finally, the synonym stage considers synonyms of words, retrieved from the WordNet lexical database. Once the matches have been found, the precision and recall are calculated by dividing the number of matches by the translation and reference length respectively. Using the precision and recall, an F_1 score is calculated and weighted by an additional penalty function to penalize incorrect word order.

Sequence Accuracy Score

A simple metric that can be used for Seq2Seq models is the sequence accuracy score. It counts how many sequences were translated correctly and divides it by the total number of sequences.

2.6.8 Conclusion

Based on this research, there are three main architectures for Seq2Seq models: recurrent, convolutional and transformer networks. Using an attention mechanism, RNN encoder-decoder model overcome the bottleneck issue of the fixed-sized context vector. However, these architectures still rely on RNNs which cannot be parallelized. With the use of CNNs it is possible to parallelize the encoding and decoding of sequences. Another approach is not to use CNNs or RNNs at all but instead make use of the attention mechanism. So-called transformer networks are currently state-of-the art for Seq2Seq models, especially in the domain of MT where their BLEU score could beat all other approaches. When building an NMT model, an important step is to select a strategy on how to handle OOV words so there are no unexpected results during testing time. For decoding, the beam search heuristic can be used to select the hypothesis that maximizes the joint probability of the target sequence for a given source sequence.

Chapter 3

Setup and Models

This chapter describes the models and the setup that were used to conduct several experiments in this project. The models were programmed in Python 3.7 with the latest version 2.0 of the TensorFlow framework (Abadi et al., 2016). The code for implementing the metrics, the input pipeline and the beam search algorithm have been inspired by the Tensorflow models repository¹. The training progress was visualized by using the TensorBoard² toolkit that provides visualization and profiling of TensorFlow programs. The models were trained and evaluated on a Quadro RTX 8000 GPU with 48GB RAM.

3.1 NOTAM Dataset

For the purpose of this project a dataset from Skyguide with more than 3.7 million NOTAMs was used. This section describes findings from the data quality assessment as well as preprocessing steps that were required to train the machine learning models.

3.1.1 Data Quality Assessment

Before evaluating several models, a data quality assessment for the provided dataset was conducted. The dataset consists of all standard properties of a NOTAM including several meta information such as a the originator of the message or the quality level. For the purpose of this project, only NOTAMs that had been processed by an AIM officer from Skyguide were analysed. All these messages have the quality level *smart*. This leaves a smaller dataset with 1.1 million entries. In addition to the original NOTAM text (item E), these NOTAMs contain also a smart version.

For further analysis, only the NOTAM text in its raw and smart version is considered and are called raw and smart NOTAM throughout this report.

Vocabulary

The raw and smart NOTAMs were tokenized. For both types, a vocabulary was built that consists of the unique tokens. The vocabulary of the raw NOTAMs contains of 427'058 tokens whereas the vocabulary of the smart NOTAMs is slightly smaller with 425'971 tokens. To measure how many words both vocabularies have in common, the Jaccard similarity (Jaccard, 1901) was calculated. A similarity of one means that the raw and smart version share the same tokens, whereas a similarity of zero indicates that there is no overlap. The similarity between the two vocabularies is 0.953, which shows that there is a high overlap. The union of the two vocabularies contains 436'901 tokens. However, 66.85% of these tokens are numbers, such as flight coordinates, frequencies or dates.

Figure 3.1 shows two word clouds for the raw and smart NOTAMs. The more frequent a word appears, the larger it is drawn. The words *RWY* and *TWY* seem to appear very frequently in both types of messages. Interestingly, the word *DUE* seems to appear often for raw NOTAMs, whereas it is not visible in the word cloud of the smart version. It is assumed that the reason for an event is often omitted in the smart version.

¹<https://github.com/tensorflow/models>, accessed 17.10.2019

²<https://www.tensorflow.org/tensorboard>, accessed 17.10.2019



Figure 3.1: Word clouds of the NOTAM messages

Length of the NOTAM Messages

The first step was to analyse the length of the messages. Several summary statistics including the quantiles 1-3 were calculated and are displayed in table 3.1. The shortest NOTAM has only a length of 1 character whereas the longest NOTAM is 4'124 characters long. Furthermore, this table shows that smart NOTAMs are in average shorter than its raw version. However, the difference is only 10 characters. Therefore it cannot be spoken of a summarization, but rather of a paraphrasing or translation task. The histogram in figure 3.2 shows the distribution of the message lengths and clearly indicates that it is right skewed.

	Length All	Length Raw NOTAM	Length Smart NOTAM
Mean	95.68	100.86	90.50
Median	50	53	48
Standard Deviation	159.48	165.19	153.39
Min	1	1	1
Max	4124	4124	4124
Quantile 1	29	31	26
Quantile 2	50	53	48
Quantile 3	107	113	104

Table 3.1: Summary statistics for the NOTAM lengths

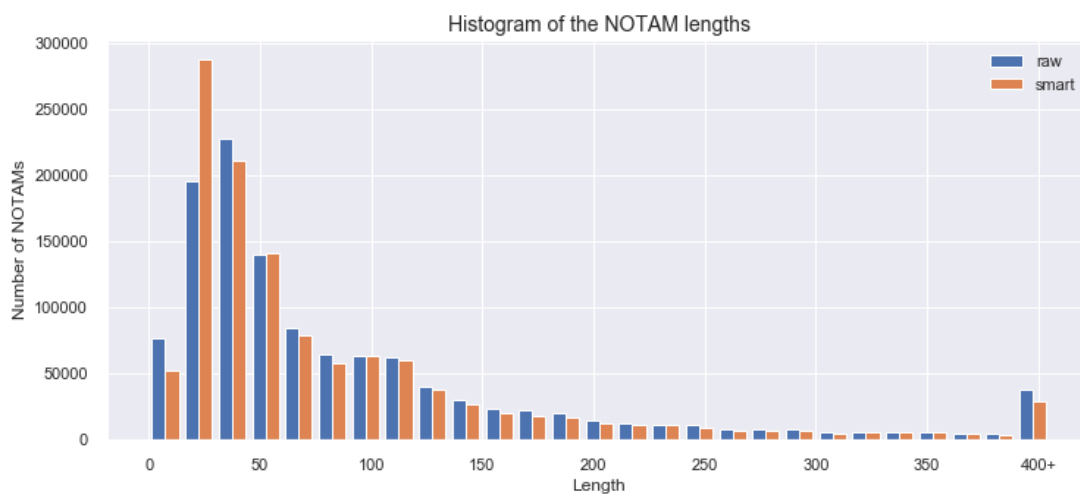


Figure 3.2: A histogram of the message lengths

Using the summary statistics, a boxplot was plotted and is shown in figure 3.3. It clearly shows that there are long messages that can be considered as outliers. The interquartile range (IQR) is the difference between the quantile 3 (Q_3) and quantile 1 (Q_1). According to the rule that items larger than $Q_1 + 1.5 \cdot \text{IQR}$ can be considered as outliers, a maximum length of 236 for raw NOTAMs and 221 for smart NOTAMs can be defined. This is an important fact that must be considered when implementing the machine learning models.



Figure 3.3: Boxplots for the NOTAM text lengths

Difference between raw and smart NOTAMs

The goal of this project is to build a system that is able to smartify NOTAMs. Thus, it is essential to know how much the two types of NOTAMs differ from each other. Figure 3.4 shows that in 49% of the cases, the length of the messages is equal after the smartification process. In 32% of the cases, the messages are shortened, i.e. information is removed or written in a more condense form. Surprisingly, in 19% of the cases more information is added: the messages are longer after being smartified.

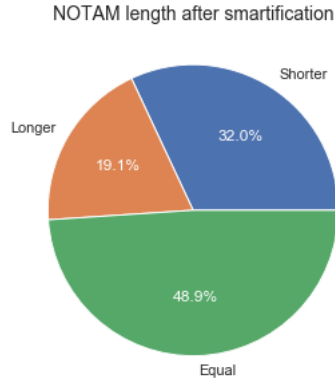


Figure 3.4: Comparison of the length after smartification

Additional analysis revealed that around 48% of the messages do not only share the same length after smartification but were not modified at all, i.e. the smart version is identical to the raw version. This is also an important observation, as it means that the dataset is imbalanced. It could happen that a smartification model gets stuck in a local optimum where it simply copies the input to the output. Even though it is useless, this model would achieve a high accuracy.

To get an understanding to what degree the messages differ from their smart version, the Levenshtein distance between the raw and smart NOTAMs was calculated. The histogram in figure 3.5 shows that raw messages leave the smartification process either unchanged or with substantial changes.

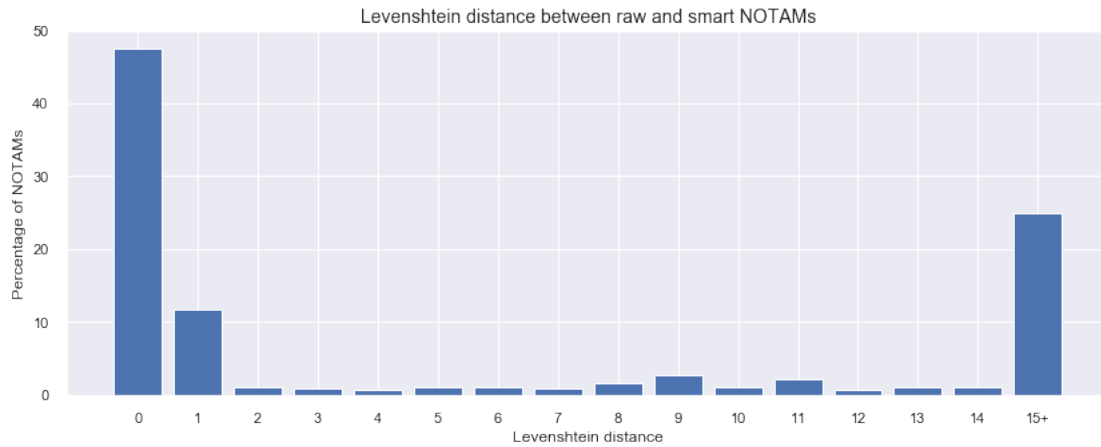


Figure 3.5: A histogram of the Levenshtein distance between the raw and the smart NOTAMs

In addition to the Levenshtein distance, the Jaccard similarity was measured. The histogram in figure 3.6 shows that the Jaccard similarity is for more than 50% of the NOTAMs equal to one.

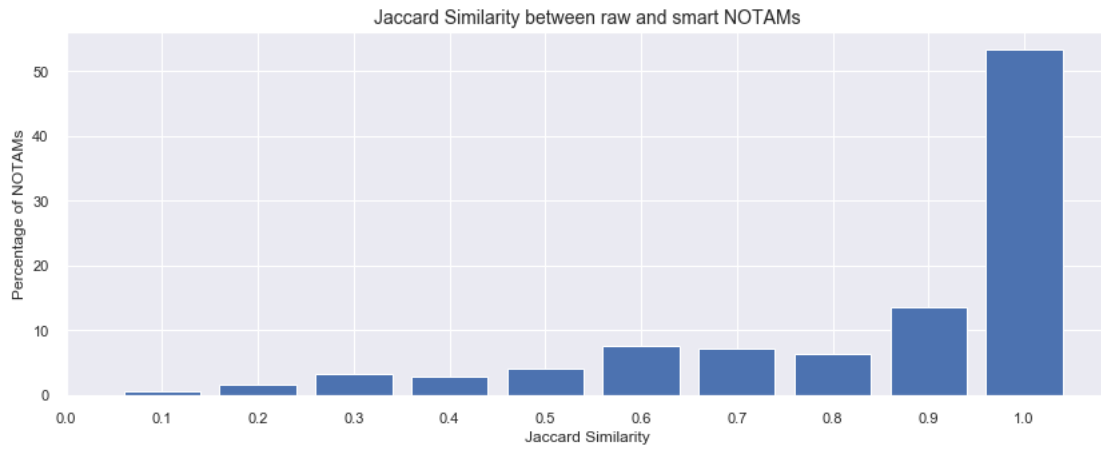


Figure 3.6: A histogram of the Jaccard similarity between the raw and the smart NOTAMs

3.1.2 Preprocessing

In order to train the machine learning models, the data needs to be preprocessed first. With the knowledge gained from the data quality assessment, the following preprocessing steps were executed.

Dataset Splitting

The full dataset is split into a training, validation and test set. According to the best practices of Ng (2017), the data in each split should be from the same distribution and have the same properties. To ensure this, the full dataset was randomly split.

Training Set The training set is used to train the machine learning models and contains 709'578 NOTAMs which is 60% of the whole dataset. The raw NOTAMs are used as input for the model and the corresponding smart NOTAMs are required to compute the loss to optimize the model.

Validation Set The validation set is used to tune the hyperparameters and select the best model. It contains 236'527 NOTAMs which is about 20% of the data.

Test Set The test set contains 236'526 NOTAMs which is again about 20% of the full dataset. It is used to evaluate the performance of the tuned model. It has to be ensured that the test set is not used until the model and its hyperparameters is selected and trained. If some data is leaked, it may happen that the evaluated performance is optimistically biased.

Vocabulary and Tokenization

For NOTAMs tables with more than 6'000 abbreviations³ exists. An interview with a private pilot from HSLU showed that it may occur that authorities invent some new abbreviations that are not official. Thus, building a fixed-sized vocabulary to train the model does not work because during testing time there might be some abbreviations that the model has not seen yet. Additionally, NOTAMs contain a lot of numbers such as coordinates or frequencies that would again result in OOV words.

As described in section 2.6.5, there are multiple techniques to handle OOV words. Backing off to a lookup dictionary (Luong et al., 2015b) is not feasible for this project because there is no universal dictionary. For this reason, a subword-unit encoder (Sennrich et al., 2015) was used, which encodes the tokens into subwords by means of the BPE algorithm. The whole dataset consists of 436'901 unique tokens, which requires a huge amount of memory if each token is one-hot encoded. With the use of a subword-unit encoder it is possible to define the number of unique tokens in the vocabulary. A vocabulary size of $2^{14} = 16'384$ unique tokens was selected and the subword-unit encoder was fitted on the training set. Reducing the vocabulary size can lead to longer training time, as the sequences will increase in length. However, experiments with an increased vocabulary size of 2^{15} and 2^{16} unique tokens did not show any improvements. Decreasing the vocabulary size to 2^{13} led to worse results, because the model is then required to learn the context of more smaller subunits.

For MT tasks it makes sense to fit two separate vocabularies: one for the source and one for the target language. However, the vocabulary for the raw and smart NOTAMs is almost identical. Thus, only one vocabulary was built. The vocabulary was extended with tokens to identify the start and the end of a NOTAM. Besides that, a padding symbol which is used to pad the sequences was introduced.

³<https://www.proairpilot.com/faa-acronyms-list.html>, accessed 17.10.2019

Figure 3.7 shows an example where the NOTAM "AIRPORT CLOSED DUE SNOW CLEARING" is first tokenized into words and then encoded into subwords. For each subword, the ID is looked up according to the index in the vocabulary.

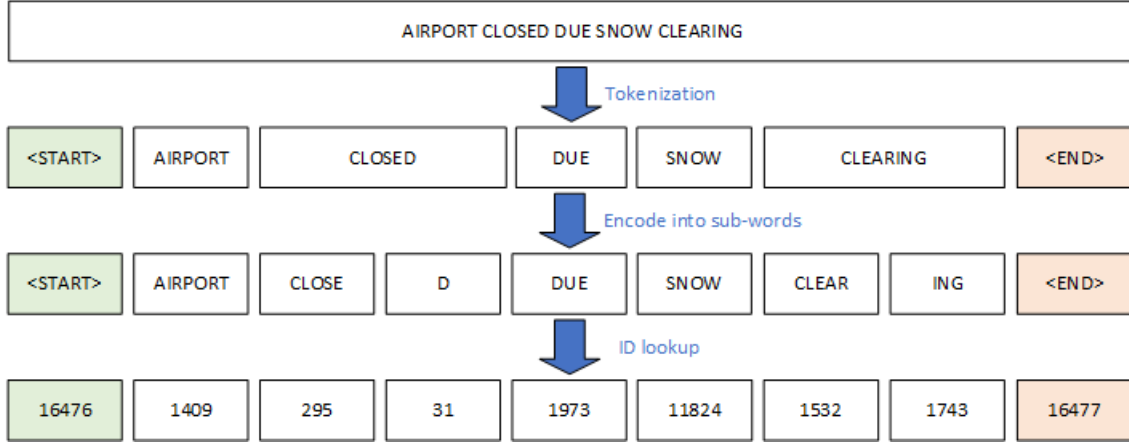


Figure 3.7: An example of the subword-unit encoder

Input Pipeline

An input pipeline that loads the tokenized dataset during the training steps was developed. ANN models have the constraint that the sequence length of each mini-batch must be identical. This can be achieved by padding all examples to a maximum length. However, the variance of the sequence length is very high. On the one hand there are NOTAMs that only contain 1 character and on the other hand there are NOTAMs with more than 4'000 characters. Padding all examples to the same length would be a waste of computation time.

A more sophisticated way is to pad the examples in a mini-batch to the maximum length of each batch. However, this is still inefficient if a mini-batch contains very short and long examples at the same time. Another disadvantage is that the fixed batch size needs to be rather small to prevent an out-of-memory error if the longest example gets selected. To allow the usage of a higher batch size and thus increase the effectiveness of training, examples larger than a maximum length could be filtered. Nevertheless, using the maximum length of 236 characters calculated in the data quality assessment would mean that about 9% of all training examples would be lost. Hence, filtering is not the best option.

To mitigate these issues, the bucketing algorithm that has been successfully used by Khomenko et al. (2017), was applied. The algorithm creates buckets for training examples that contain sentences of similar length. The batch size is not fix, but varies according to the current samples. In order to fully utilize the provided hardware, the maximum tokens per batch was set to 10'000. With the used training set, this results in a batch size of around 306 NOTAMs in average.

Figure 3.8 shows an example where the data is arranged into buckets according to their length.

Bucket 1	<START>	AIRPORT	CLOSE	D	DUE	SNOW	CLEAR	ING	<END>
	<START>	R	AREA	LI-	R315	ACT	<END>	<PAD>	<PAD>
				...					
Bucket 2	<START>	EXP	DLY	.	<END>				
	<START>	AIRSPACE	CLSD	<END>	<PAD>				
	<START>	RWY	23	LOC	<END>				
				...					

Figure 3.8: An example of the bucketing algorithm

3.2 Classification Models

The data quality assessment revealed that almost 48% of the NOTAMs are already in its smart version, i.e. no further processing step is required. A model that predicts whether a NOTAM needs to be smartified is valuable for Skyguide as their AIM officers would save much time. Consequently, it makes sense to implement a text classification model that fulfils this goal. The remainder of this section describes the text classification models that were implemented and evaluated for this project.

3.2.1 Data Preprocessing

To train the text classification models, binary labels were generated for each NOTAM. If a message and its smart version are identical, it is labelled as zero, i.e. it is already smart. If they differ, the message is labelled as one, i.e. it needs to be smartified.

3.2.2 Metrics

To compare the different models, the classification metrics accuracy, precision, recall and F_1 score were measured. For the task of classifying whether a NOTAM needs to be smartified or not, a high recall is preferred over a high precision because it is unfortunate to miss NOTAMs that should have been smartified. During the smartification process the messages are rephrased and standardised. Missing a NOTAM is therefore only critical if further information is added. The data analysis showed that this is the case for 19.1% of the NOTAMs.

3.2.3 Baseline Classification Model

In the conducted data quality assessment it has been noticed, that a lot of NOTAMs are smartified by removing the reason for the message and adding a period-symbol (.) at the end of the text. For example the message "AIRPORT CLOSED DUE SNOW CLEARING" is smartified to "AP CLSD.". In detail, 40.37% of all raw NOTAMs end with a period whereas it is 70.84% for smart NOTAMs. In addition, 10.36% of the raw messages contain the token *DUE*, while only 1.94% of the smartified messages contain it.

With these straightforward statistics, a rule-based text classification model was implemented that classifies all NOTAMs as to be changed (*prediction* = 1) if it contains the token "DUE" or does not end with a period.

3.2.4 Recurrent Classification Model

A classification model based on RNNs was implemented. The model consists of two bidirectional stacked GRU layers. The output of the RNN is fed into a linear layer. The sigmoid function is applied to obtain the probabilities that a certain NOTAM needs to be changed or not. To stabilize the gradients, layer normalization is applied between the layers.

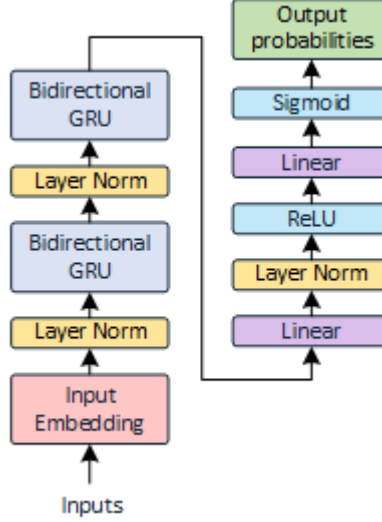


Figure 3.9: The computation graph of the RNN classification model

Table 3.2 shows the setup of the recurrent classification model that was tuned based on experimentation on the validation set. It turned out that using LSTM layers instead of GRU layers could not improve the performance. Moreover, stacking two bidirectional layers seemed to be enough to capture long-distance dependencies. Adding more layers led to overfitting and thus reduced the performance measured on the validation set. To improve the training speed and to stabilize the gradients, layer normalization has been applied between the layers. Instead of using the activation of the last hidden layer, the attention mechanism could be used to allow the model to pay attention to previous timesteps. However, this resulted in a model that did not converge.

Setting	Value
Embeddings Size	512
RNN	
GRU units	[512, 512]
GRU activation	tanh
MLP	
MLP units	[1024, 1]
Hidden layer activation	ReLU

Table 3.2: Setup of the RNN model

3.2.5 Convolutional Classification Model

Inspired by the proposed architecture of Jacovi et al. (2018), a text classification model based on CNNs was implemented. Figure 3.10 shows the architecture of the network. After mapping the input into the embedding space, multiple 1-dimensional convolution layers with different filter sizes are applied in parallel. For each filter output, the maximum along the sequence length is taken which results in a matrix of dimensions (batch size \times filter size). These outputs are concatenated and fed into a MLP to calculate the output probabilities.

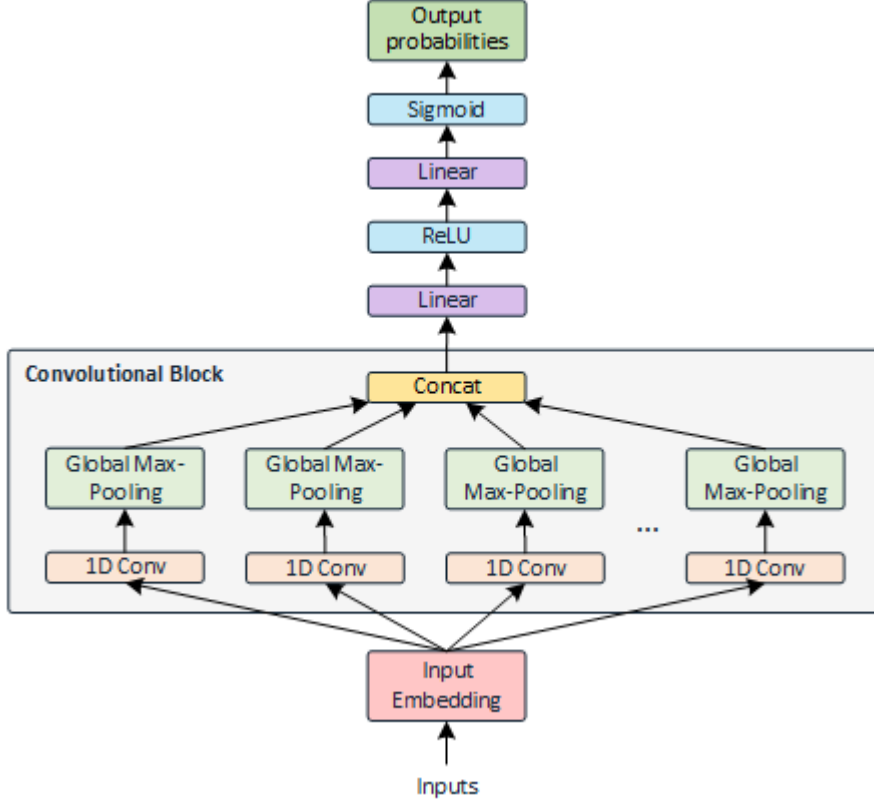


Figure 3.10: The CNN model used for NOTAM classification

The corresponding settings for the model are given in table 3.3 which were determined based on the performance measured on the validation set of several experiments. The model uses 5 different convolutional layers in parallel. Each layer acts as an n -gram detector. For example a filter of size 2 considers bigrams, a filter of size 3 trigrams etc. All convolutional layers use 200 filters in parallel.

Setting	Value
Embeddings Size	512
CNN	
Filter sizes	[2, 3, 4, 5, 6]
Number of filters	[200, 200, 200, 200, 200]
Strides	[1, 1, 1, 1, 1]
Activation	ReLU
MLP	
MLP units	[1024, 1]
Hidden layer activation	ReLU

Table 3.3: Setup of the CNN model

3.2.6 Combination of CNN and RNN

Based on the work of Wang et al. (2016) described in section 2.5.3, the RNN and CNN models were combined as depicted in figure 3.11. An embedding vector is obtained for each input and then fed into multiple 1-dimensional convolutional filters in parallel. Again, these convolutional layer have different filter sizes to capture different features. Max-pooling is applied on these features to get the most significant ones. These obtained features are concatenated and fed into a bidirectional GRU layer. The GRU layer is used to obtain a fixed-sized vector from the CNN output. Finally, the last hidden state of the GRU layer is applied in a linear layer with a sigmoid activation function to obtain the output probabilities.

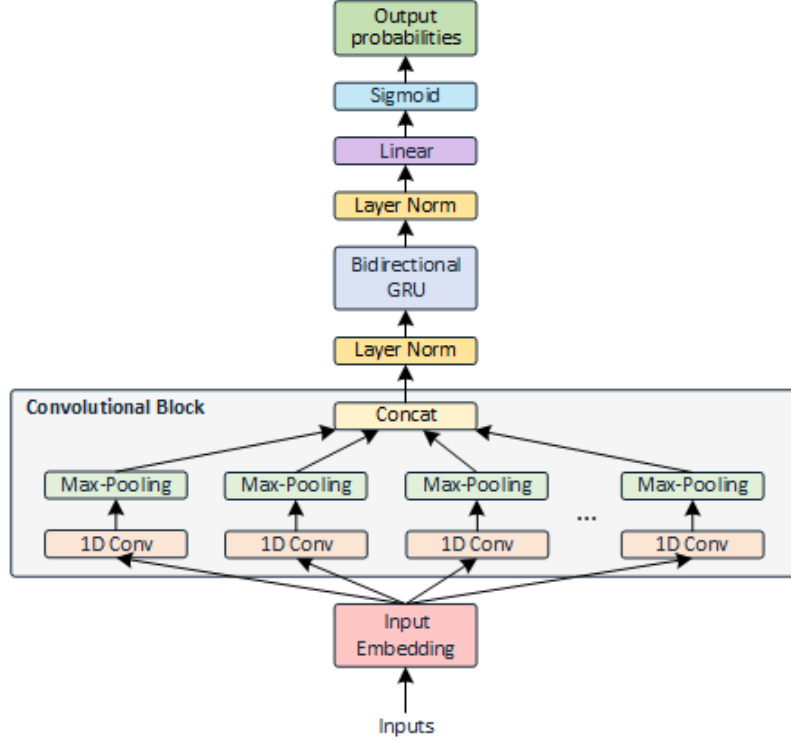


Figure 3.11: Combination of CNN and RNN architecture

Table 3.4 shows the setup of the model. In comparison to the CNN-based model described in section 3.2.5, max-pooling with stride 2 is applied for each convolution filter rather than global-max pooling. This does not return a fixed-sized vector, but a vector with variable length in respect to the input length. The RNN uses only one GRU cell with 1'024 units.

Setting	Value
Embeddings Size	512
CNN	
Filter sizes	[2, 3, 4, 5, 6]
Number of filters	[200, 200, 200, 200, 200]
Strides	[1, 1, 1, 1, 1]
Activation	ReLU
Max-pooling strides	[2, 2, 2, 2, 2]
RNN	
GRU units	[1024]
MLP	
MLP units	[1024, 1]
Hidden layer activation	ReLU

Table 3.4: Setup of the hybrid classification model

3.2.7 Transformer Classification Model

Based on the transformer for Seq2Seq tasks which was proposed by Vaswani et al. (2017), a classification model was implemented. The output of the transformer encoder is a matrix of shape (batch size \times input sequence length \times model dimension). In order to use a MLP to calculate the output probabilities, the encoder output needs to have a constant size. Similar to the CNN model, global max-pooling is applied over the sequence length to obtain an output shape (batch size \times model dimension). Alternatively the output could be averaged. However, global max-pooling showed to achieve a better result in comparison to calculating the mean.

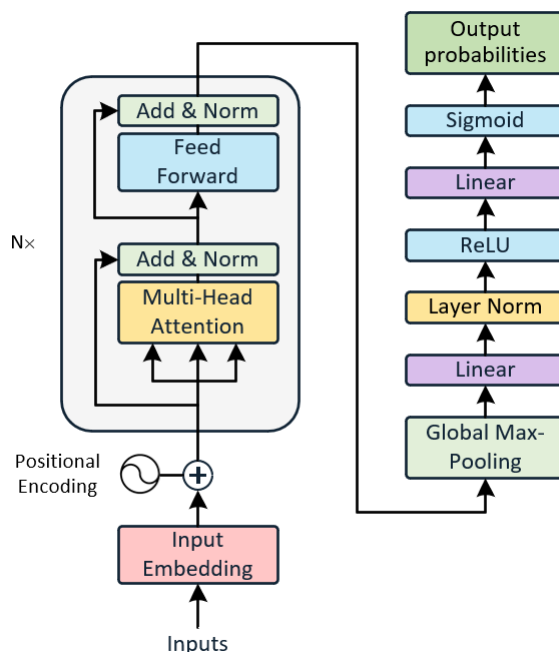


Figure 3.12: The architecture of the transformer classification model that uses the encoder of the original transformer model proposed by Vaswani et al. (2017)

The settings for the transformer model are given in table 3.5. The authors of the original transformer model proposed to stack 6 layers and to use a dimension of 512 for the model. Moreover, they used 2’048 units for the feed forward layers. Using these settings, the model was not able to learn and ended up in a local minium where it always returned the prediction 1. By reducing the model capacity, the model was able to learn the mapping.

Setting	Value
Transformer	
Number of layers	4
Number of heads	8
Dimension feed forward	512
Dimension model	128
Feed forward activation	ReLU
MLP	
MLP units	[1024, 1]
Hidden layer activation	ReLU

Table 3.5: Setup of the transformer classification model

3.2.8 Optimizer Settings

The aforementioned models were trained using the adam optimizer with the default settings $\beta_1 = 0.9$ and $\epsilon = 10^{-7}$. The hyperparameters for the optimizer are given in table 3.6. To decrease training time, a scheduler for the parameter β_2 was developed that linearly increases its value from 0.5 to the default 0.999 over a total of 50'000 training steps. This has the consequence that the effective learning rate starts with a larger number and thus leads to larger training steps. The number of warmup-steps was determined based on experimentation on the validation set. Figure 3.13 shows a plot of the implemented β_2 scheduler.

Setting	Value
Optimizer	Adam
Decay rate β_1	0.9
Decay rate β_2	Scheduler
Learning rate	10^{-4}
Loss function	Binary Cross Entropy

Table 3.6: Optimzier configuration for the text classification models

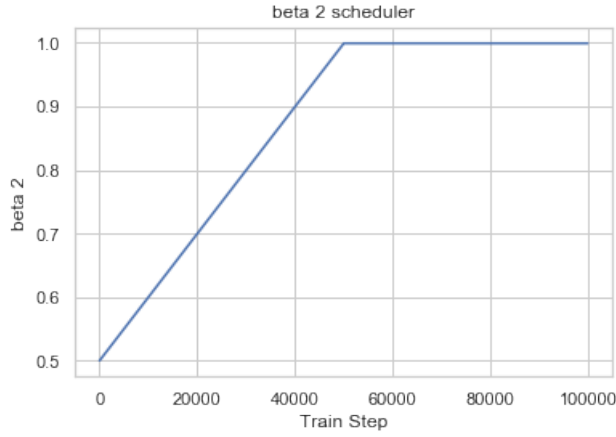


Figure 3.13: A plot of the β_2 scheduler

3.3 Sequence to Sequence Models

This section describes the Seq2Seq models that were implemented and evaluated in this project.

3.3.1 Metrics

As described in section 2.6.7, there are several metrics for evaluating Seq2Seq models. The most common metric for MT tasks is the BLEU score and for text summarization the ROUGE score. From the conducted data quality assessment it is not clear whether the transformation of a NOTAM into a smart NOTAM can be regarded as a summarization process or a translation process as there are some cases where the smart version is longer than the raw one. Therefore both BLEU and ROUGE scores were tracked. The WER metric, described in section 2.6.7, was not used because its computation is very inefficient. Furthermore, the METEOR score is not suitable for the NOTAM smartification process as it requires access to a stemmer and lexical database.

During the training process, the following metrics are recorded using the TensorBoard visualization toolkit. To decrease training time, the metrics on the validation set were only computed every 5 epochs.

BLEU Score The computation of the $BLEU_N$ score was parametrized with the suggested parameters from Papineni et al. (2002): $N = 4$ and uniform weights $w_n = 1/N$. During the training, the BLEU score is computed for each mini-batch and then averaged after each epoch. However, this score is not the original corpus BLEU score but a sentence-level BLEU score. Because of its rather complex computation, it is only calculated on the validation set after each epoch.

ROUGE Score The $ROUGE_n$ F_1 score has been parametrized with $n = 2$, i.e. to compute the score, the number of bigram overlaps is considered. It is computed for each training step and averaged by the end of an epoch. Like the BLEU score it is only computed on the validation set.

Accuracy Score The accuracy score has the advantage that its calculation is rather light-weight in comparison to the BLEU or ROUGE score. Thus, it has been computed on the training and validation set. However, it is not used for the model selection.

Top- k Accuracy Score The top- k accuracy marks a token as correct if it is contained in the top- k probabilities. This score gives a good intuition whether the training process is headed onto the right direction. It was tracked on the training and validation set. The score was parametrized with $k = 5$. Like the accuracy score, it is not used for model selection.

Sequence Accuracy Score The sequence accuracy score marks a sequence as correct if every token is identical to the ground truth. For the model evaluation it is being tracked on the training and validation set.

To train the Seq2Seq models, the teacher-forcing method is applied where the true output is passed to the next time step regardless of what the model predicts at the current time step. This is a very efficient way of training, however it can lead to optimistically biased metrics. Consequently, a separate evaluation script was implemented that does not use the teacher-forcing method to produce its output. By means of the evaluation script not only the sentence BLEU score was computed, but also the corpus BLEU score. To check if the performance can be improved by applying the beam search algorithm, the metrics are recorded for multiple beam sizes.

3.3.2 Baseline Seq2Seq Model

To get an understanding how the metrics can be interpreted, the following baseline model has been implemented: the model outputs the same data as the input, i.e. it is assumed that the input NOTAM is already smart. Relatively high values for the metrics can be expected as the data quality assessment showed that the Levenshtein distance is very small for a high number of samples.

3.3.3 Transformer Seq2Seq Model

For the task of smartifying NOTAMs, a transformer Seq2Seq model was implemented. The implementation of the model is based on a tutorial by TensorFlow which is available on their website ⁴. The tutorial explains how a NMT model can be built to translate Portuguese sentences into English. The following sub-section discusses different transformer model configurations that were evaluated for the task of smartifying NOTAMs.

Transformer Pointer-Generator Model

NOTAMs contain many factual details such as coordinates, flight numbers or frequencies which pose a problem for classic Seq2Seq models. The accurate generation of such numbers is crucial for the smartification process because wrong numbers would change the complete meaning of the messages. If for example a frequency X is out of service, it is necessary that the frequency is copied one-to-one from the source sentence and is not changed in the slightest way. If that is not the case, this would introduce high security risks for pilots.

In most of the cases, these detailed numbers are contained in the source sentence and do not need to be generated. Hence, it makes sense to extend the existing transformer generator model with a *pointer*, that allows the model to copy certain information directly from the source to the output text. The implementation of the pointer-network extension is inspired by a public GitHub repository ⁵, where a transformer pointer-generator is implemented with TensorFlow 1.x in the context of text summarization.

Figure 3.14 shows the computation graph of the model. To compute the generation probabilities \mathbf{p}_{gen} , the decoder input \mathbf{x}_t , the corresponding decoder output \mathbf{s}_t and the context vector \mathbf{c}_t are concatenated and fed into a pointer-network. The pointer-network in equation 3.1 is a shallow fully-connected network with one hidden layer. The result is fed into a sigmoid activation to calculate the output \mathbf{p}_{gen} , which is a vector that indicates the probability for each token to be generated or not. In contrast to the implementation of Deaton et al. (2018), the attention weights α are obtained after the second multi-head attention layer in the last decoder layer by taking the average of all heads instead of summing them up. The context vector \mathbf{c}_t is taken as the sum across the source input of the final encoder layer's output, weighted by the attention weights α .

$$\mathbf{p}_{\text{gen}} = \sigma\left(\text{pointer_nw}([\mathbf{x}_t; \mathbf{s}_t; \mathbf{c}_t])\right) \quad (3.1)$$

However, the computation of the context vector \mathbf{c}_t has shown to be expensive and requiring a lot of memory. To train the whole pointer-generator network, the maximum number of tokens per batch had to be reduced from 10'000 to 3'000 which increased the training time. Consequently, a second model configuration using only the decoder input \mathbf{x}_t and the decoder output \mathbf{s}_t was implemented. Equation 3.2 shows how the generation probabilities are calculated with the second configuration.

$$\mathbf{p}_{\text{gen}} = \sigma\left(\text{pointer_nw}([\mathbf{x}_t; \mathbf{s}_t])\right) \quad (3.2)$$

⁴<https://www.tensorflow.org/tutorials/text/transformer>, accessed 17.10.2019

⁵<https://github.com/policeme/transformer-pointer-generator>, accessed 13.11.2019

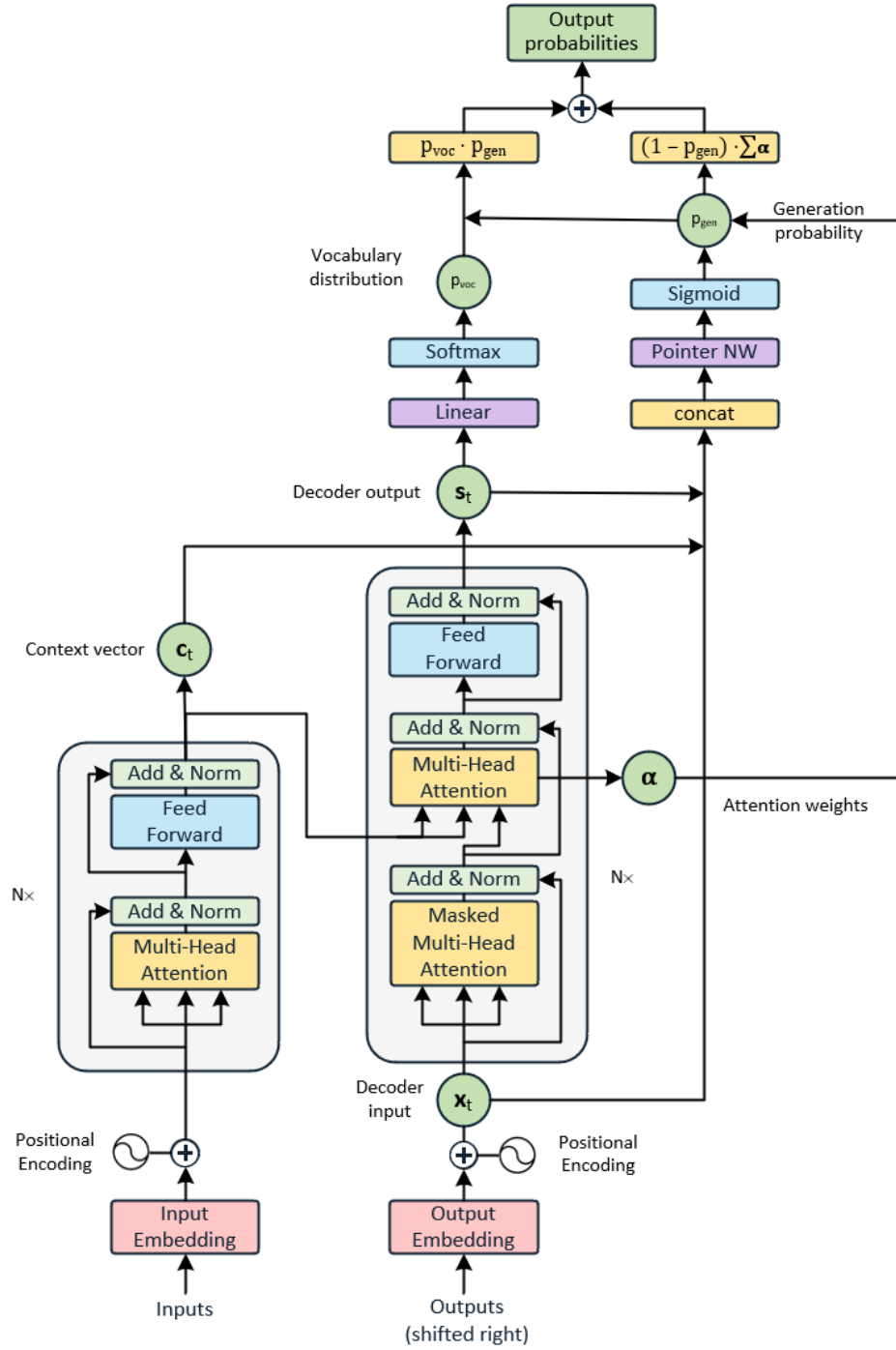


Figure 3.14: The pointer-generator transformer model for the NOTAM smartification process

Model Parameters

Table 3.7 shows the setup of the transformer model. Vaswani et al. (2017) proposed to use 6 encoder and decoder layers each. When using such a deep network, however, a poor performance was achieved. Thus the transformer model was parametrized with only 4 layers. The dimension of the feed forward layers was also reduced from 2'048 to 1'024 units. These settings lead to better performance and prevented the model from overfitting.

Transformer-based state-of-the-art NLP models such as BERT from Devlin et al. (2018) propose to use the GELU instead of the ReLU activation function for the feed forward layers. Nevertheless, no improve in the performance or training speed could be observed. The MLP network for the pointer-generator has been parametrized by one hidden layer with 1'024 units. The ReLU activation function was used of the hidden layer of the pointer-network. While training the pointer-generator network, some numerical instabilities occurred which resulted in the loss becoming undefined. To prevent these instabilities, an error term $\epsilon_{p_{\text{gen}}}$ was introduced that is added to the generation probabilities p_{gen} if they become less than a tolerance of 10^{-15} . Subsequently the error term is subtracted in case the probability is one. This theoretically introduces a small bias, but experimentation have shown that this does not affect the performance of the model. The last layer of the network is a fully-connected layer with 16'384 units, which is the same number as the size of the vocabulary.

Setting	Value
Transformer	
Number of layers	4
Number of heads	8
Dimension feed forward	1024
Dimension model	512
Feed forward activation	ReLU
Pointer-network settings	
MLP units	[1024, 1]
Hidden activation	ReLU
Error term $\epsilon_{p_{\text{gen}}}$	10^{-11}
Last layer of the generator	
linear layer units	16'384 (vocab size)

Table 3.7: Setup of the transformer classification model

Coverage Loss

A coverage loss that has been introduced by See et al. (2017) and described in section 2.6.2 was implemented with the goal to reduce repetitions in the output. By adding the coverage loss to the total loss, the model is penalized for repeatedly paying attention to the same tokens during decoding time. However, introducing a coverage loss did not improve the performance. This could be due to the fact that it is mainly used for text summarization where the output is significantly shorter than the input. In the domain of NOTAM smartification, the difference between the output and the input length is rather small. Manual inspection of the generated output showed that errors due to repetition are rather rare.

Optimizer

To train the transformer model, the adam optimizer with the parameters shown in table 3.8 was used. The categorical cross entropy loss function is applied to calculate the loss during training time.

Setting	Value
Optimizer	Adam
Decay rate β_1	0.9
Decay rate β_2	0.98
Learning rate	Scheduler
Loss function	Categorical Cross Entropy

Table 3.8: Optimzier configuration of the transformer Seq2Seq model

Furthermore, as proposed by Vaswani et al. (2017), a learning rate scheduler was implemented that varies the learning rate according to equation 3.3. The learning rate is linearly increased during a warmup period and then decayed over time. The parameter d_{model} is the dimension of the model. Figure 3.15 shows a plot of the learning rate scheduler parametrized with a model dimension of 512 and 4'000 warmup steps.

$$\text{learning_rate} = d_{\text{model}}^{-0.5} \cdot \min(\text{step_num}^{-0.5}, \text{step_num} \cdot \text{warmup_steps}^{-1.5}) \quad (3.3)$$

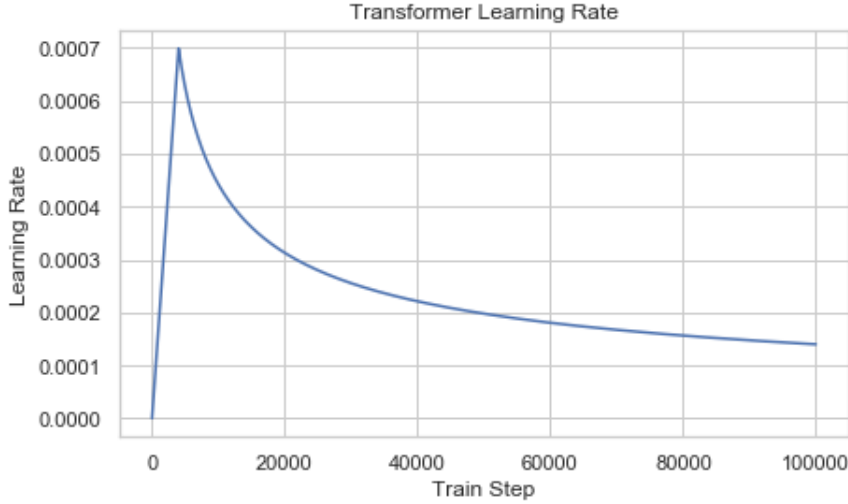


Figure 3.15: The learning rate scheduler of the transformer model

Regularization

To prevent overfitting, Vaswani et al. (2017) used label smoothing for the transformer model. Label smoothing is a regularization technique for classification problems to prevent the model from predicting the labels too confidently during training and thus generalizes poorly. It has been introduced by Szegedy et al. (2015) in the context of image classification. With label smoothing the one-hot encoded labels are modified according to equation 3.4 where K is the number of classes and ϵ_{ls} the label smoothing factor. Instead of minimizing the cross-entropy cost function with hard targets, the model minimizes it by using the soft targets. For the NOTAM smartification process, the label smoothing approach was used with the parameter $\epsilon_{\text{ls}} = 0.1$.

$$y_k^{\text{ls}} = y_k(1 - \epsilon_{\text{ls}}) + \frac{\epsilon_{\text{ls}}}{K} \quad (3.4)$$

In addition to label smoothing, dropout with a rate of 0.1 is applied to the output of each sub-layer.

3.3.4 Hierarchical Model

Assuming the text classification model achieves a high accuracy in predicting whether the messages need to be changed or not, it would make sense to combine it with the smartification model. If a NOTAM is classified as already smart, it does not need to be processed by the Seq2Seq model. This way not only computation time could be saved but also the accuracy in generating the output could be increased.

For this reasons, a hierarchical model that combines the classification and the Seq2Seq model was implemented. A NOTAM is first fed into the text classification model that decides if the NOTAM needs to be smartified or not. If not, the original NOTAM is returned as in the baseline model. Otherwise it is fed into the Seq2Seq model to perform the smartification. Figure 3.16 shows the activity diagram of this hierarchical approach.

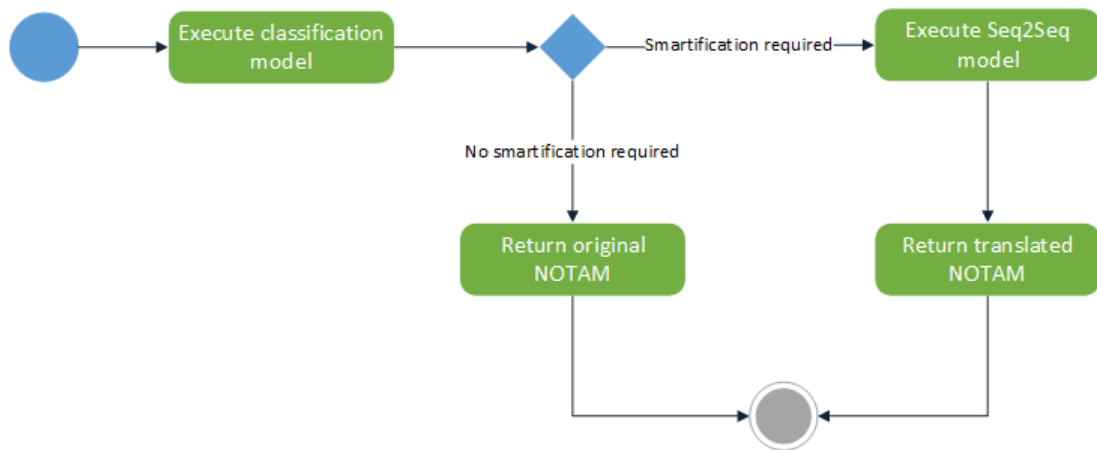


Figure 3.16: The activity diagram of the hierarchical model

Chapter 4

Results

This chapter presents the results of the trained models for the tasks of classifying whether a NOTAM needs to be changed or not and for the task of smartifying NOTAMs.

4.1 NOTAM Classification

In this section the results of the different classification models explained in the previous chapter are discussed.

4.1.1 Model Comparison

Hyperparameter tuning was conducted for all implemented model architectures. The models were trained with the selected hyperparameters described in the previous chapter for a total of 50 epochs. After training, the metrics were calculated on the validation set and are displayed in table 4.1. The following section compares the results of the baseline model and the different model architectures.

Baseline The baseline model that classifies a NOTAM as to be changed if it contains the token *DUE* or does not end with a period.

RNN The classification model that is based on RNNs.

CNN The model that uses multiple CNN filters to extract features for classification.

CNN + RNN The hybrid classification model that uses a combination between the CNN and RNN model.

Transformer The model that uses the transformer encoder to encode the input into a vector that is used for classification.

Metrics

Table 4.1 shows that all models perform better than the baseline model which achieves an accuracy of almost 67%. The transformer model performs the worst with an F_1 score of 89.20%. Nevertheless, the difference between the other models is not significant in terms of F_1 score.

Model	Accuracy	Precision	Recall	F_1 Score
Baseline	66.64%	65.10%	78.40%	71.13%
RNN	92.83%	93.65%	92.61%	93.13%
CNN	93.28%	92.67%	94.67%	93.66%
CNN + RNN	93.39%	93.15%	94.33%	93.74%
Transformer	88.91%	91.11%	87.38%	89.20%

Table 4.1: The metrics of the classification models calculated on the validation set

Figure 4.1 shows the receiver operating characteristic (ROC) curve where the TPR is plotted against the FPR at various threshold settings. The diagonal line represents a random classifier. The further the line of a classifier is from the diagonal, the better it is. The figure on the right shows the ROC curve zoomed in at top left.

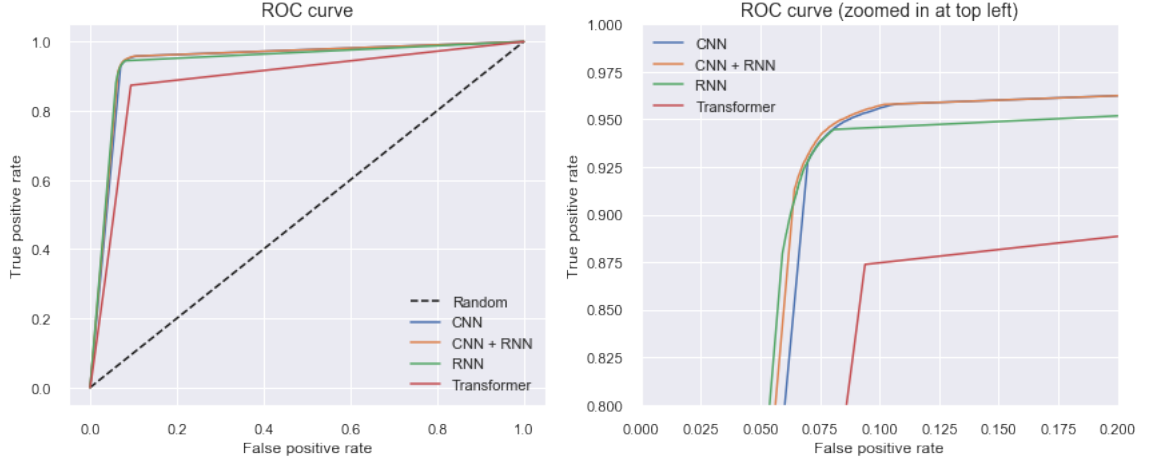


Figure 4.1: ROC curve of the different classification models.

It shows that all models performs clearly better than random guess. The zoomed plot on the right shows that the transformer model performs the worst. There is no significant difference between the other classifiers visible.

Comparison of Parameters

Table 4.2 compares the models according to the number of parameters, training time and average inference time. The classification model based on CNNs shows to be the fastest in inference time, even though it consists of 11.5 Mio. parameters. The model which combines CNNs and RNNs is the slowest in inference time. The transformer model takes only 1.5 hours to train 50 epochs, whereas the RNN needs 7.5 hours.

Model	No. params	Training Time	Avg. Inference Time
RNN	17.4 Mio.	≈ 7.5 h	≈ 0.120 ms
CNN	11.5 Mio.	≈ 2 h	≈ 0.093 ms
CNN + RNN	18 Mio.	≈ 7 h	≈ 0.270 ms
Transformer	3 Mio.	≈ 1.5 h	≈ 0.179 ms

Table 4.2: Comparison of the classification models in terms of parameters, training time and inference time

Model Selection

For final evaluation, the CNN-based model has been selected. It is not only the fastest in inference time but also achieves the highest recall score of 94.67% on the validation set. With a training time of approximately 2 hours, it is almost as fast as the transformer version.

4.1.2 Final Evaluation

To get an understanding how the model performs in production, it was evaluated on unseen test data. Table 4.3 shows the metrics of the CNN model computed on the test set. The results are almost identical to the one calculated on the validation set.

Accuracy	Precision	Recall	F ₁ Score
93.27%	92.70%	94.64%	93.66%

Table 4.3: The metrics of the CNN-based classification model calculated on the test set

Figure 4.2 shows the corresponding confusion matrix. There are in total 6'659 false positives: these NOTAMs were classified as already smart ($prediction = 0$) even though they should be smartified.

		Predicted	
		True	False
Actual	True	117657	6659
	False	9264	102945

Figure 4.2: The confusion matrix of the classification model evaluated on the test set

Classifying a NOTAM as smart even though it would require smartification is bad. In cases where the Levenshtein distance between the raw and smart version is exceptionally large, it is even worse. To get a feeling in how many cases this occurs, the metrics were calculated on two additional subsets: for NOTAMs where the Levenshtein distance is smaller than 2 and for NOTAMs where the levenshtein distance is greater or equal to 15. Table 4.4 shows the metrics achieved on these subsets. The corresponding confusion matrices are shown in figure 4.3.

The precision for the first subset is with 74% relatively low. This can also be seen in the confusion matrix where there are 9'279 false positives. However, classifying them as to be changed is not a huge problem if the smartification process is done manually. The number of false negatives is with 962 items relatively low and is not such a serious problem because a pilot would not recognize that these NOTAMs were not smartified. The second subset does not have any false positives, since all messages have to be smartified. Thus the precision is 100%. The false negatives for the second subset are the most critical ones: these NOTAMs would require a lot of processing, but they were recognized as already smart. There are 2'389 of such critical examples from a total of 236'526 NOTAMs which is about 1% of the test set.

Levenshtein	Examples	Accuracy	Precision	Recall	F ₁ Score
< 2	139'954	92.68%	74.27%	96.53%	83.95%
≥ 15	58'862	94.91%	100%	94.91%	97.39%

Table 4.4: The metrics of the CNN-based model compared to the levenshtein distance.

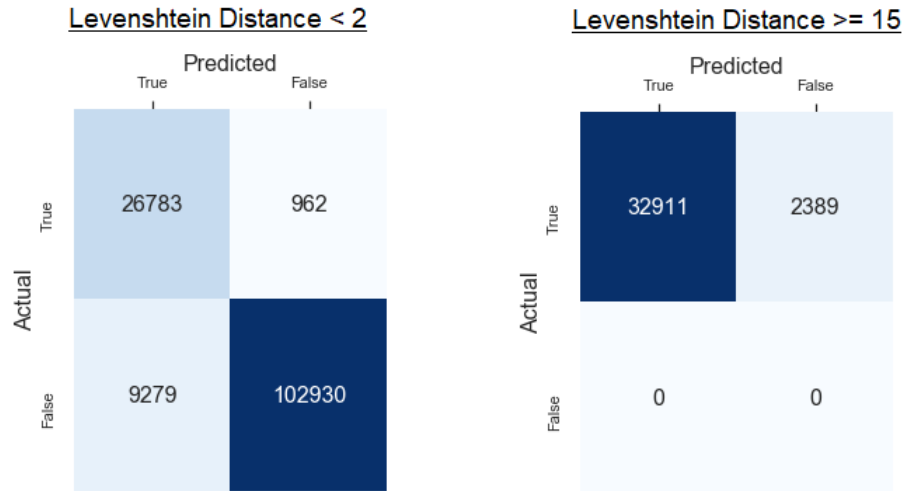


Figure 4.3: The confusion matrix of the two subsets

To check whether the model fails to classify long messages correctly, the correlation between the length of the messages and the misclassification rate has been calculated. With a pearson correlation of only 0.0059, the length and misclassification rate do not show any significant correlation.

4.2 NOTAM Smartification

This chapter presents the results for different configurations of the transformer smartification model. A configuration is selected and then evaluated on unseen test data.

4.2.1 Model Comparison

The following section compares the results of the baseline model and the different transformer model configurations. All models were trained with the hyperparameters described in the previous chapter for a total of 50 epochs.

Baseline The baseline model that copies the raw NOTAM.

(A) Transformer The original transformer model that does not use the pointer-network.

(B) Transformer pointer-gen. The transformer pointer-generator model described in the previous chapter.

(C) Transformer pointer-gen. (no context) The transformer pointer-generator model that does not use the context vector \mathbf{c}_t to calculate the generation probabilities \mathbf{p}_{gen} .

The metrics were calculated on three validation subsets: the subset *All* contains the whole validation set whereas the subset *Not changed* contains only the NOTAMs that were not changed during the smartification process. The third subset *Changed* contains the NOTAMs that are being modified during smartification.

Metrics

The results of the baseline model that simply returns the raw NOTAM are shown in table 4.5. The model achieves a corpus-level BLEU₄ score of 76.20% on the whole validation set, which is already a very high score. Contrarily, the sequence accuracy is only 47.57% which reflects the percentage of not changed NOTAMs that has been detected in the data quality assessment. The scores for the *Not Changed* subset are of course 100% for all metrics.

Model	Validation Subset	BLEU ₄ Score (Sentence-lvl)	BLEU ₄ Score (Corpus-lvl)	ROUGE ₂ F ₁ Score	Sequence Accuracy
Baseline	All	73.27%	76.20%	77.64%	47.57%
	Not changed	100%	100%	100%	100%
	Changed	49.02%	51.62%	57.35%	0%

Table 4.5: The results of the NOTAM smartification baseline model measured on the validation set

Table 4.6 shows the metrics of the trained models evaluated on the validation set. The best result per metric, model and subset is written in bold. The model configuration (A) achieves the best performance in most cases. However, these differences are not significant and based on these results it is not possible to select a best performing model.

Model	Validation Subset	BLEU ₄ Score (Sentence-lvl)	BLEU ₄ Score (Corpus-lvl)	ROUGE ₂ F ₁ Score	Sequence Accuracy
(A) Transformer	All	87.69%	86.37%	90.40%	69.64%
	Not changed	97.19%	97.95%	97.90%	92.77%
	Changed	79.07%	72.21%	83.60%	48.66%
(B) Transformer pointer-gen.	All	87.62%	86.47%	90.34%	69.60%
	Not changed	97.18%	97.82%	97.89%	92.75%
	Changed	78.94%	72.30%	83.49%	48.61%
(C) Transformer pointer-gen. (no context)	All	87.47%	86.03%	90.19%	69.25%
	Not changed	97.19%	97.99%	97.87%	92.93%
	Changed	78.65%	71.44%	83.23%	47.77%

Table 4.6: The results of the NOTAM smartification models trained on the the whole training set and measured on the validation set

Factual Details

As mentioned earlier, generating factual details correctly is crucial for the smartification process. To check whether the models are able to generate the factual details correctly, all tokens except those containing at least one number were removed from the validation set and the model outputs. For example the text *"ILS 'MM' RWY02 FREQ 75MHZ U/S."* is transformed into *"RWY02 75MHZ"*. Based on these labels and the corresponding model output, the BLEU score and the accuracy was calculated. For averaging the accuracy score, the micro weighting was used, i.e. the number of correct tokens are summed up and divided by the total number of tokens. Moreover, the Jaccard similarity for each example was calculated and averaged.

Model	BLEU ₄ (Corpus-lvl)	Accuracy	Jaccard Similarity
Baseline	86.58%	87.27%	86.72%
(A) Transformer	92.46%	90.33%	94.50%
(B) Transformer pointer-gen.	92.19%	89.14%	94.20%
(C) Transformer pointer-gen. (no context)	92.16%	90.00%	94.28%

Table 4.7: Metrics calculated only on numerical tokens

Table 4.7 shows that there are no significant differences between the three transformer models. The BLEU score is even higher if it is calculated on numerical tokens only. The accuracy score is about 90%, meaning that in 10% of the cases the factual details were either missed or not in the correct position. The relatively high Jaccard similarity of about 94% means that only 6% of the numbers are either incorrect or missing. Surprisingly, the Jaccard similarity for the baseline model is not 100%, which means that not all numbers are already contained in the raw NOTAM.

Comparison of Pointer-Generator Networks

Due to the fact that the performance of the transformer network with and without the pointer-network extension is almost identical, it could be assumed that the model learns to generate the whole output, i.e. that the pointer-network has no effect. To test this hypothesis, the pointer-generator models were evaluated in two different manners: by only using the pointer-network and by only using the generator. If the pointer-generator only uses the pointer-network, all resulting tokens would be copied from the source NOTAM. It can be assumed that the performance would then be similar to the one of the baseline model. If the pointer-generator only uses the generator, the output would be generated using not only the vocabulary of the input NOTAM, but also the whole vocabulary. If the assumption holds true, the pointer-network would perform as good as the baseline model whereas the generator would perform almost as good as when using the full model.

Model	Validation Subset	BLEU ₄ Score (Sentence-lvl)	BLEU ₄ Score (Corpus-lvl)	ROUGE ₂ F ₁ Score	Sequence Accuracy
(B) pointer	All	75.22%	78.59%	79.29%	48.30%
	Not changed	99.49%	99.57%	99.67%	98.20%
	Changed	53.21%	55.28%	60.80%	3.03%
(B) generator	All	12.53%	6.08%	15.3%	2.54%
	Not changed	3.99%	1.34%	3.88%	0.04%
	Changed	20.29%	10.85%	25.66%	4.82%
(C) pointer	All	75.09%	78.29%	79.20%	48.22%
	Not changed	99.53%	99.65%	99.69%	98.27%
	Changed	52.92%	54.82%	60.61%	2.80%
(C) generator	All	12.7%	6.96%	15.42%	2.54%
	Not changed	3.92%	1.52%	3.64%	0.06%
	Changed	20.68%	12.35%	26.11%	4.80%

Table 4.8: The results of the pointer-generator models with either using only the pointer or only the generator

Table 4.8 shows that the hypothesis can be rejected: the pointer-network has a huge impact on the performance of the pointer-generator network. The performance of the pointer-network is even better than the baseline model which means that the model has learnt not only to copy the whole input sequence but also to reorder words. Manual inspection of the results revealed that reordering the words is indeed sometimes the case. For example, the NOTAM *"ILS RWY 28L GP U / S"* is smartified to *"RWY 28L ILS GP U / S"* by only using the pointer-network. However, the correct smart version would be *"RWY 28L ILS GP U/S."* The pointer-network is of course not able to do that because it cannot add additional tokens or replace them. Contrarily, the performance when only using the generator is very bad. However, in combination with the pointer-network it contributes to the final result.

Interestingly, this decomposition reflects how the final result is constructed. For example, if the sentence-level BLEU score of the pointer and generator are added together, the result is almost identical to the one of the original pointer-generator model. This assumption does not hold true for the sequence accuracy. Again, both pointer-generator networks achieve almost the same results. Therefore it can be assumed that the context information \mathbf{c}_t is either not required to calculate the generation probabilities \mathbf{p}_{gen} or the information is already contained in the decoder input \mathbf{x}_t .

Comparison of Parameters

Table 4.9 compares the number of parameters and training time of the different model configurations. Adding the pointer-network (B) to the transformer model increases the number of parameters by 1.5 million. The configuration (C) has 0.4 million fewer parameters. The model (A) has the shortest training time with 16 hours for a total of 50 epochs. Due to memory issues, the configuration (B) can only be trained with maximum 3'000 tokens per batch while the other configurations are trained with 10'000 tokens per batch. Hence, the configuration (B) takes almost twice as long to train.

Model	No. params	Training Time
(A) Transformer	46.4 Mio.	≈ 16 h
(B) Transformer pointer gen.	47.9 Mio.	≈ 30 h
(C) Transformer pointer gen. (no context)	47.5 Mio	≈ 22 h

Table 4.9: Comparison of the Seq2Seq models in terms of parameters and training time

Model Selection

Even though the performance of the transformer model could not be improved by extending it with a pointer-network, it has still some interesting properties. First of all, its decision can be better interpreted. Not only the attention weights provide information about the decision but also the probability that a token was copied during prediction. This gives an intuition on how the model came to its decision while generating the output. Figure 4.4 shows an example where the NOTAM *"TWY B BTN TWY W AND TXL B7 CLOSED DUE WIP."* is transformed to *"TWY B BTN TWY W AND TXL B7 CLSD."* The attention weights plot shows to which tokens the model had to pay attention to, whereas the bar plot of the copy probabilities indicates which tokens the model has copied directly from the source sequence.

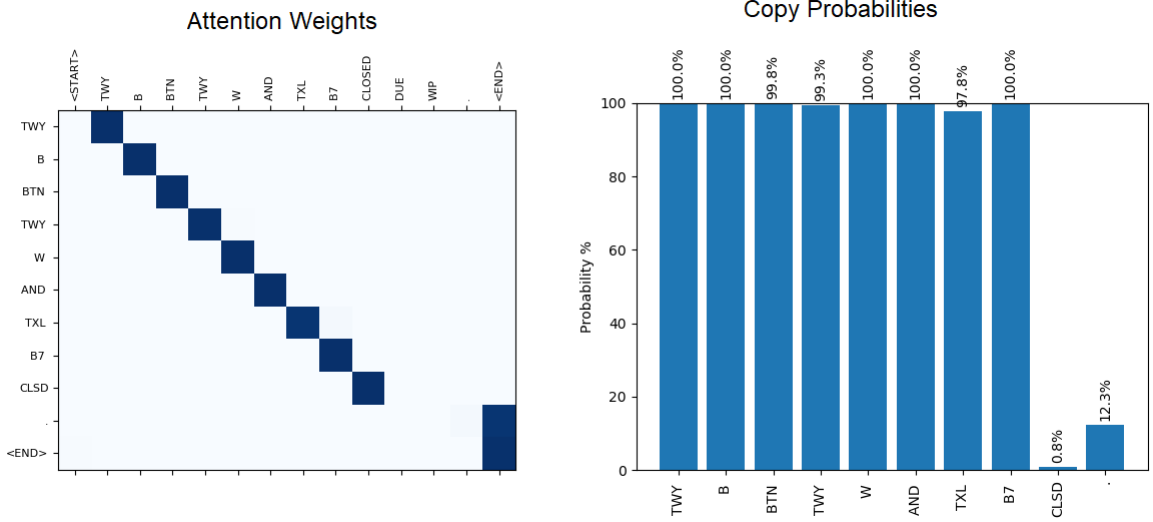


Figure 4.4: An example of attention weights and copy probabilities for a smartified NOTAM

Another interesting property of pointer-generator networks is that they are better in handling OOV words. If future NOTAMs contain new abbreviations that the model has never seen, the pointer-network allows the model to copy them more easily from the source text.

Because of better interpretation properties as well as the easier way of handling OOV words, the transformer model with the pointer-network extension was selected for final evaluation. A comparison showed that the usage of the context vector \mathbf{c}_t is not required, thus the model configuration (C) was selected. This configuration is easier to train than model (B) because it is faster and requires less memory.

4.2.2 Hierarchical Model

To check whether the performance of the smartification task can be improved by using the proposed hierarchical approach, the metrics were calculated on the validation set. The hierarchical model was parametrized with the selected CNN-based classification model and with the transformer pointer-generator Seq2Seq model (C). Table 4.10 shows the metrics and further compares the performance of the hierarchical model if the Seq2Seq model is trained using only the changed NOTAMs. Surprisingly, the performance on the *Changed* validation subset could only be slightly improved when training it only on changed NOTAMs. Overall, combining the classification and Seq2Seq model did not result in better performance. Hence, the hierarchical model is not further considered.

Training Data	Validation Subset	BLEU ₄ Score (Sentence-lvl)	BLEU ₄ Score (Corpus-lvl)	ROUGE ₂ F ₁ Score	Sequence Accuracy
All	All	87.53%	86.08%	90.22%	69.60%
	Not changed	97.75	98.49%	98.32%	94.38%
	Changed	78.25%	71.11%	82.86%	47.12%
Changed	All	86.89%	85.97%	89.70%	68.29%
	Not changed	96.45%	96.89%	97.24%	91.94%
	Changed	78.22%	72.04%	82.86%	46.83%

Table 4.10: The results of the NOTAM smartification hierarchical model measured on the validation set.

4.2.3 Final Evaluation

The pointer-generator model configuration (C) was evaluated on unseen test data. The results in table 4.11 do not show significant differences to the metrics calculated on the validation set.

Test Subset	BLEU ₄ Score (Sentence-lvl)	BLEU ₄ Score (Corpus-lvl)	ROUGE ₂ F ₁ Score	Sequence Accuracy
All	87.51%	86.12%	90.23%	69.28%
Not changed	97.20%	97.97%	97.87%	92.87%
Changed	78.77%	71.56%	83.32%	47.99%

Table 4.11: The results of the NOTAM smartification model measured on the test set

To get a deeper understanding for what kind of NOTAMs the model works well, several more evaluations were conducted and are presented in the remainder of this sub-section.

Beam Search

To evaluate the performance of the model, the beam search algorithm was implemented according to the formula from Wu et al. (2016) (see section 2.6.6) with the proposed setting $\alpha = 0.6$. However, no coverage penalty factor was used. The results were recorded for the beam sizes 1 (greedy search), 2 and 3 and 5.

Beam Size	Test Subset	BLEU ₄ Score (Sentence-Level)	BLEU ₄ Score (Corpus-Level)	ROUGE ₂ F ₁ Score	Sequence Accuracy
1 (greedy)	All	87.51%	86.12%	90.23%	69.28%
	Not changed	97.20%	97.97%	97.87%	92.87%
	Changed	78.77%	71.56%	83.32%	47.99%
2	All	85.32%	82.53%	88.60%	62.12%
	Not changed	97.17%	93.95%	97.87%	93.34%
	Changed	74.63%	68.59%	80.22%	33.95%
3	All	84.71%	82.11%	88.13%	60.83%
	Not changed	96.87%	93.88%	97.65%	92.19%
	Changed	73.73%	67.76%	79.54%	32.53%
5	All	84.14%	81.73%	87.66%	60.02%
	Not changed	96.76%	93.84%	97.56%	91.73%
	Changed	72.76%	66.96%	78.72%	31.39%

Table 4.12: The result of the NOTAM smartification model for different beam sizes

Table 4.12 reveals the metrics for different beam sizes. Surprisingly, only the sequence accuracy could be slightly improved on the *Not Changed* subset when using beam size 2. For the other beam sizes no improve could be reported. Thus it can be concluded that greedy search, which takes the token with the highest probability at each step, is best suited for generating NOTAMs.

NOTAM Length

To check whether there is a correlation between the raw NOTAM length and the performance measures, the pearson correlation was calculated. Both, the ROUGE score and BLEU have a slight negative correlation of -0.13 with the NOTAM length, i.e. the longer a NOTAM is, the harder it is to predict correctly. The correlation between the sequence accuracy and the length is with -0.17 even higher. However, this makes sense as the chance of making a mistake increases with increasing sequence length.

Hard Examples

There are in total 169 messages where the model achieved a BLEU score of 1% or less. Manual inspection on these messages showed that the smartified version often contains additional information that cannot simply be derived from the raw message. Table 4.13 shows examples of NOTAMs where the model failed to smartify them.

In the first example, additional information is added that is not contained in the raw NOTAM. It is very likely that the AIM officer from Skyguide queried these data in some other data sources. The second example shows a NOTAM that is completely written in Spanish. The smart version looks like a standard NOTAM with the use of proper abbreviations. Because only few Spanish examples are available, the model did not learn to translate it into a smart NOTAM. The NOTAM in the last example is again very interesting because the text is written in both English and Italian. Even though it can be expected that the corresponding smart version should be only English, nothing was changed. It can be assumed that this example was missed by a Skyguide employee. When feeding it to the model, it is translated as follows: *"LDG, TKOF AND TAX WITH CTN DUE TO GRASS CUTTING."* Even though the model output looks correct for humans, it achieves a pretty low BLEU score, because only few n -grams are contained in the ground truth. Surprisingly, the model was able to skip the whole Italian part and only smartified the English one.

Raw NOTAM	Smart NOTAM (Ground Truth)
MCTR ACTIVATED. ENTRY COND ACCORDING TO AIP AUSTRIA LOXZ AD 2.3	09 1130-1648, 10 0800-1649, 11 0800-1651, 12 1530-1652, 13 1530-1654, 14 1530-1656, 15 1530-1657, 16 1130-1659, 17 0800-1700, 18 0800-1702, 19 1530-1703, 20 1530-1705, 21 1530-1706, 22 1530-1708 ZELTWEG MCTR ACT.
HORAS TRANQUILAS POR CEREMONIA MILITAR. LLEGADAS SALIDAS, RODAJES, PRACTICAS DE APROXIMACION, ENCENDIDO DE MOTORES NO AUTORIZADOS EXCEPTO AERONAVES INCLUIDAS EN LA CEREMONIA Y EMERGENCIAS. TODAS ACFT PODRIAN SUFRIR DEMORAS EN ESTE TIEMPO	DEP AND ARR, SHOOTING ACFT AND APCH PRACTICES NOT AUTH EXC~ EMERG. EXP DLA.
TESTO ITALIANO: ATTERRAGGI, DECOLLI E RULLAGGI CON PRECAUZIONE CAUSA SFALCIO ERBA. UOMINI E MEZZI IN CONTATTO RADIO CON LA LOCALE UNITA' DI SERVIZIO INFORMAZIONI VOLO (AFIU)	TESTO ITALIANO: ATTERRAGGI, DECOLLI E RULLAGGI CON PRECAUZIONE CAUSA SFALCIO ERBA. UOMINI E MEZZI IN CONTATTO RADIO CON LA LOCALE UNITA' DI SERVIZIO INFORMAZIONI VOLO (AFIU)
ENGLISH TEXT: LANDING, TAKE-OFF AND TAXI WITH CAUTION DUE TO GRASS CUTTING. MEN AND EQPT IN RADIO CONTACT WITH LOCAL AERODROME FLIGHT INFORMATION UNIT (AFIU) REF AIP AD 2 LILE 1-1	ENGLISH TEXT: LANDING, TAKE-OFF AND TAXI WITH CAUTION DUE TO GRASS CUTTING. MEN AND EQPT IN RADIO CONTACT WITH LOCAL AERODROME FLIGHT INFORMATION UNIT (AFIU) REF AIP AD 2 LILE 1-1

Table 4.13: Examples of NOTAMs where the model failed to smartify them

Chapter 5

Discussion

Chapter 5 discusses the results of this project and provides an outlook regarding future work.

5.1 Outlook

There are several things that could be investigated in order to improve the current work. Moreover, based on the findings of this project, some further research could be done in the domain of NOTAMs.

5.1.1 Possible Improvements

The following section describes some possible improvements to increase the performance of the text classification and smartification models.

Detect NOTAMs in a Foreign Language

In the data quality assessment, it was not recognized that there are NOTAMs written in another language than English. The preprocessing steps should be extended to detect such NOTAMs. However, simple language detection does not work because an example containing only abbreviations is likely to be detected as a language other than English. It makes sense to train the classification model using these examples, since all these messages have to be smartified. On the other hand, training the smartification model with these NOTAMs seems pointless, because it is highly unlikely that the model is able to learn a foreign language with only few examples.

Feature Engineering for Text Classification Model

The text classification model only uses the actual NOTAM text as a feature. The dataset from Skyguide consists of other features such as the originator of the original NOTAM. Conducting some feature engineering for the classification task could result in an increased performance of the model. If the classification accuracy could be increased, the use of the hierarchical smartification model could be interesting again.

Reuse Embeddings

The NOTAM smartification models contain two embedding layers: one for the encoder and one for the decoder. Due to the fact that the vocabularies of the raw and smart NOTAMs are almost identical, it would make sense to share these embeddings. This would lead to fewer parameters and it is highly likely that the model would require shorter training time. In addition to that, the embeddings could be pre-trained by either using a word embeddings model such as FastText, or a language model like BERT. This way the training time could be reduced requiring less number of epochs to train.

Handle Digits

The data quality assessment showed that over 66% of the unique tokens of the vocabulary consists of numerical characters. The current solution does not treat numerical and alphabetical characters separately. In most cases the numerical tokens are already contained in the source sequence. To achieve that such tokens are not modified in the slightest way, all tokens could be masked during preprocessing to indicate whether it contains a number or not. In cases where the generator outputs a numerical token, they could be penalized, which would then result in the pointer-network having more weight. However, as there was no literature found for cases like this, more investigation would be required. Alternatively, the vocabulary could be built without numerical characters. During decoding time, numbers would then be treated as OOV words. By dynamically extending the vocabulary, such tokens could be copied using the pointer-network.

Implement Confidence Metric

The implemented smartification model does not yield a confidence value for its output. Rikters and Fishel (2017) proposed to use the attention distributions as a confidence metric in the context of RNN-based NMT models. The authors claim that good translations can be characterized by strongly focused attention connections. However, they exhibited only weak correlation with human judgements. For the purpose of the NOTAM smartification model, another confidence metric needs to be developed because a transformer pointer-generator model was used rather than an RNN-based Seq2Seq model. The implementation of a quality level for the model output could give the Skyguide employees a quick indicator to decide whether the output is trustworthy or requires manual revision.

5.1.2 Extend Smart NOTAM Workflow

In this project, the pipeline of the whole NOTAM smartification workflow has been extended by the NOTAM text classification and the NOTAM text prediction. The last step of the workflow is the prediction of the Q-code. In a future work, the evaluated Seq2Seq model for the task of NOTAM text prediction could be used to train a model that is able to generate the Q-code of a NOTAM.

5.1.3 NOTAM Translation

The smartification process transforms the NOTAMs into a standardised phraseology. Even though these are standardised abbreviations, they are still hard to understand for inexperienced people. Another interesting research area would be the translation of NOTAMs into proper English. Because NOTAMs are broadcasted publicly, there are millions of examples available which could be used to train machine learning models. Since no English translations are available for the NOTAMs, supervised Seq2Seq models such as those used for the smartification tasks cannot be applied. Instead unsupervised or semi-supervised models need to be developed.

5.2 Conclusion

At the moment, the smartification of NOTAM messages is completely made by Skyguide AIM officers without assistance of intelligent algorithms. From the thousands of messages broadcasted every year, approximately 50% of them need to be processed. Due to the high amount of manual work, the question arises as to whether the smartification can be fully or at least partially automated.

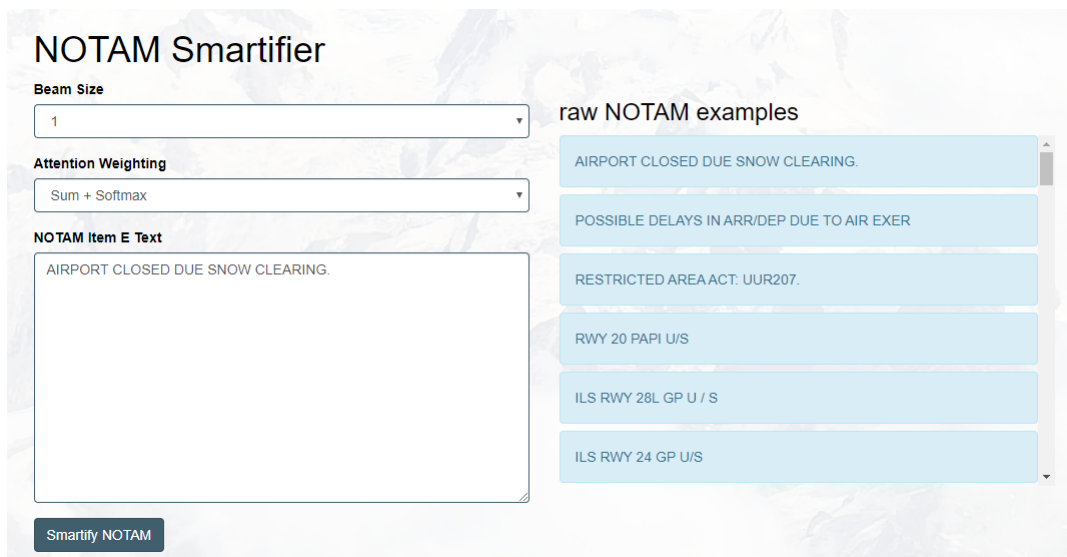
This research project ascertains that the use of machine learning algorithms for the smartification workflow is clearly possible. Nevertheless, there is a zero error tolerance in the domain of aviation. Any changes or omissions of some factual details by the smartification algorithm would pose a serious safety risk to pilots. Even though the scores of the evaluated model measured on unseen test data are exceptionally high, there are still some cases where it is unable to perform the smartification. For example if some factual information is added or if the raw NOTAM is written in a foreign language. Therefore, fully automating the smartification workflow without human supervision seems not feasible at the moment. However, the industrialisation of a semi-automatic process, in which the algorithm suggests a possible smart NOTAM and is then checked by a human, is possible. Such a process is still a great advantage for Skyguide as in most of the cases the output is correct.

Appendix A

Appendix

A.1 Demonstration App

A demonstration app that allows the user to try out the evaluated smartification and classification model was implemented. The web-app is available at **notam.abiz.ch** and can be accessed with the password **HSLU-VM01**. It allows the user to enter a NOTAM item E text or alternatively select one from a list. Additionally, the beam size for decoding can be selected. Figure A.1 shows a screenshot of the app with an example input.



The screenshot shows the 'NOTAM Smartifier' web application. On the left, there are three input fields: 'Beam Size' with a dropdown menu showing '1', 'Attention Weighting' with a dropdown menu showing 'Sum + Softmax', and 'NOTAM Item E Text' with a text area containing 'AIRPORT CLOSED DUE SNOW CLEARING.'. Below these fields is a 'Smartify NOTAM' button. On the right, there is a section titled 'raw NOTAM examples' with a list of six NOTAM items in light blue boxes: 'AIRPORT CLOSED DUE SNOW CLEARING.', 'POSSIBLE DELAYS IN ARR/DEP DUE TO AIR EXER', 'RESTRICTED AREA ACT: UUR207.', 'RWY 20 PAPI U/S', 'ILS RWY 28L GP U / S', and 'ILS RWY 24 GP U/S'.

Figure A.1: An example input for the smartification web-app

The entered NOTAM is tokenized, fed into the classification model and afterwards into the smartification model. The classification model outputs the probability that the NOTAM needs to be smartified. The smartification model performs the smartification process and returns the attention distributions over the multiple heads. To visualize only one attention distribution instead of one per head, the following weighting options are available:

Mean Calculates the mean attention over all heads of the transformer model.

Sum + Softmax Sums up the attention over all heads and applies the softmax function.

Max Takes the maximum attention per head.

Moreover, a barplot of the copy probabilities from the pointer-network are displayed in a plot.

In figure A.2 the output of the web-app is displayed.

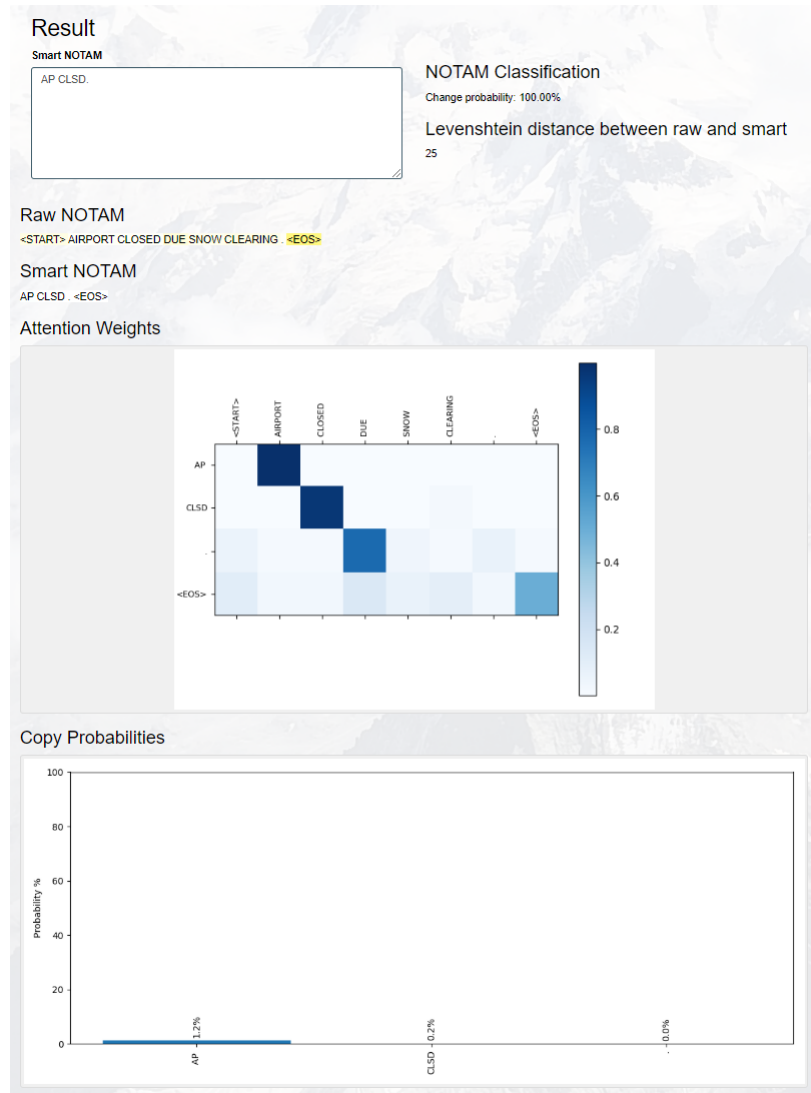


Figure A.2: An example input of the smartification web-app

A.2 Code

The following section should give a brief overview over the implemented code and how the models can be trained and evaluated.

A.2.1 Jupyter Notebooks

The following Jupyter notebooks were implemented:

- 00 Data Quality Assessment.ipynb** To generate all plots and statistics that were described in section 3.1.1.
- 01 Preprocessing.ipynb** Data preprocessing steps such as filtering and tokenization.
- 02a Evaluation Classification.ipynb** To generate the evaluation output for the classification model described in section 4.1.
- 02b Evaluation Seq2Seq.ipynb** To generate the evaluation output for the Seq2Seq model described in section 4.2.

03 Plots.ipynb Notebook to generate several plots such as the learning rate scheduler of the transformer model.

A.2.2 NOTAM Classification Model

To train the NOTAM classification model, the script *train_classification.py* was implemented that starts the training process for the classification and the Seq2Seq model respectively. The script can be executed as follows:

```
$ python3 train_classification.py
  --data_dir="/app/data/classification"
  --data_dir="/app/work"
  --max_tokens_per_batch=10000
  --run_name="classification"
```

After training the classification model, the script *eval_classification.py* can be executed to calculate the metrics.

```
$ python3 eval_classification.py
  --data_dir="/app/data/classification"
  --dataset="val"
  --max_tokens_per_batch=10000
  --checkpoint_dir="/app/work/checkpoint/classification"
```

A.2.3 NOTAM Smartification Model

To train the Seq2Seq models, the following script can be executed:

```
$ python3 train_nmt.py
  --data_dir="/app/data/preprocessed"
  --work_dir="/app/work"
  --max_tokens_per_batch=3000
  --run_name="smartification"
```

When the training is done, the models can be evaluated by running the following script:

```
$ python3 eval_nmt.py
  --data_dir="/app/data/classification"
  --dataset="val"
  --max_tokens_per_batch=10000
  --checkpoint_dir="/app/work/checkpoint/classification"
  --beam_size=1
```

List of Figures

1.1	The workflow and project scope of the smart NOTAMs	2
2.1	A graphical illustration of a MP neuron proposed by McCulloch and Pitts (1943)	4
2.2	Perceptron	5
2.3	An example of a MLP with one hidden layer and two inputs	6
2.4	The sigmoid activation function	6
2.5	The tanh activation function	7
2.6	The ReLU activation function	7
2.7	The leaky ReLU activation function with $\alpha = 0.1$	8
2.8	The ELU activation function with $\alpha = 1$	9
2.9	The GELU activation function	9
2.10	A visualization of a residual block by He et al. (2015)	13
2.11	Visualization of a simple RNN cell by Colah (2015)	14
2.12	Visualization of a GRU cell by Colah (2015)	15
2.13	Visualization of a LSTM cell by Colah (2015)	15
2.14	A CNN layer with size (3×3) , stride = 2 and padding = 1 applied to an input of dimension (5×5)	17
2.15	An example of max-pooling and average-pooling with a pooling size (2×2) and stride 2	18
2.16	Visualization of the CBOW model by Rong (2014)	19
2.17	In illustration of the extraction of the flair embedding for the word <i>Washington</i> from the original flair paper by Akbik et al. (2018)	21
2.18	An example on how to extract features for text classification using an RNN	23
2.19	A text classification example using CNNs	24
2.20	An example on how to extract features for text classification using CNNs	25
2.21	The confusion matrix	26
2.22	An encoder-decoder model	29
2.23	An encoder-decoder model with attention	29
2.24	An example of an alignment matrix	30
2.25	An example of a encoder-decoder network with attention to generate the next token . .	31
2.26	An example of a pointer-generator network	33
2.27	The architecture of the original transformer network proposed by Vaswani et al. (2017)	35
2.28	A visualization of the computation graph of the multi-head attention in Vaswani et al. (2017)	37
2.29	An example of a beam search decoder with beam size 2	39
3.1	Word clouds of the NOTAM messages	45
3.2	A histogram of the message lengths	45
3.3	Boxplots for the NOTAM text lengths	46
3.4	Comparison of the length after smartification	46
3.5	A histogram of the Levenshtein distance between the raw and the smart NOTAMs . . .	47
3.6	A histogram of the Jaccard similarity between the raw and the smart NOTAMs	47
3.7	An example of the subword-unit encoder	49
3.8	An example of the bucketing algorithm	50
3.9	The computation graph of the RNN classification model	52
3.10	The CNN model used for NOTAM classification	53
3.11	Combination of CNN and RNN architecture	54
3.12	The architecture of the transformer classification model that uses the encoder of the original transformer model proposed by Vaswani et al. (2017)	55
3.13	A plot of the β_2 scheduler	56
3.14	The pointer-generator transformer model for the NOTAM smartification process	59

LIST OF FIGURES

3.15	The learning rate scheduler of the transformer model	61
3.16	The activity diagram of the hierarchical model	62
4.1	ROC curve of the different classification models.	64
4.2	The confusion matrix of the classification model evaluated on the test set	65
4.3	The confusion matrix of the two subsets	66
4.4	An example of attention weights and copy probabilities for a smartified NOTAM	70
A.1	An example input for the smartification web-app	75
A.2	An example input of the smartification web-app	76

List of Tables

1.1	Examples of NOTAMs with their smartified version	2
2.1	BLEU score example for unigram precision	41
3.1	Summary statistics for the NOTAM lengths	45
3.2	Setup of the RNN model	52
3.3	Setup of the CNN model	53
3.4	Setup of the hybrid classification model	54
3.5	Setup of the transformer classification model	55
3.6	Optimzier configuration for the text classification models	56
3.7	Setup of the transformer classification model	60
3.8	Optimzier configuration of the transformer Seq2Seq model	61
4.1	The metrics of the classification models calculated on the validation set	63
4.2	Comparison of the classification models in terms of parameters, training time and inference time	64
4.3	The metrics of the CNN-based classification model calculated on the test set	65
4.4	The metrics of the CNN-based model compared to the levenshtein distance.	65
4.5	The results of the NOTAM smartification baseline model measured on the validation set	67
4.6	The results of the NOTAM smartification models trained on the the whole training set and measured on the validation set	67
4.7	Metrics calulcated only on numerical tokens	68
4.8	The results of the pointer-generator models with either using only the pointer or only the generator	68
4.9	Comparison of the Seq2Seq models in terms of parameters and training time	69
4.10	The results of the NOTAM smartification hierarchical model measured on the validation set.	70
4.11	The results of the NOTAM smartification model measured on the test set	71
4.12	The result of the NOTAM smartification model for different beam sizes	71
4.13	Examples of NOTAMs where the model failed to smartify them	72

List of Algorithms

1	Gradient descent algorithm	10
2	Momentum algorithm	11
3	RMSProp algorithm	11
4	Adam algorithm	11
5	Batch normalization	12
6	Layer normalization	12
7	Compute the next hidden state \mathbf{h}_t for a GRU cell	15
8	Compute the next hidden state \mathbf{h}_t for a LSTM cell	16

Bibliography

- Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems.
- Akbik, A., Blythe, D., and Vollgraf, R. (2018). Contextual string embeddings for sequence labeling. In *COLING*.
- Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization.
- Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate.
- Banerjee, S. and Lavie, A. (2005). METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.
- Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146.
- Botev, A., Lever, G., and Barber, D. (2016). Nesterov’s accelerated gradient and momentum as approximations to regularised update descent.
- 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.
- Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus).
- Colah (2015). Understanding lstm networks.
- Conneau, A., Schwenk, H., Barrault, L., and Lecun, Y. (2017). Very deep convolutional networks for text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 1107–1116, Valencia, Spain. Association for Computational Linguistics.
- Deaton, J., Jacobs, A., and Kenealy, K. (2018). Transformer and pointer-generator networks for abstractive summarization.
- Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding.
- Doddington, G. (2002). Automatic evaluation of machine translation quality using n-gram co-occurrence statistics. In *Proceedings of the Second International Conference on Human Language Technology Research, HLT ’02*, pages 138–145, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- Elman, J. L. (1990). Finding structure in time. *COGNITIVE SCIENCE*, 14(2):179–211.
- Ganesan, K. (2018). Rouge 2.0: Updated and improved measures for evaluation of summarization tasks.

BIBLIOGRAPHY

- Gehring, J., Auli, M., Grangier, D., and Dauphin, Y. N. (2016). A convolutional encoder model for neural machine translation.
- Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. (2017). Convolutional sequence to sequence learning.
- Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In Gordon, G., Dunson, D., and Dudík, M., editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA. PMLR.
- Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. The MIT Press.
- Gulcehre, C., Ahn, S., Nallapati, R., Zhou, B., and Bengio, Y. (2016). Pointing the unknown words.
- He, H. and Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Trans. on Knowl. and Data Eng.*, 21(9):1263–1284.
- He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition.
- Hendrycks, D. and Gimpel, K. (2016). Gaussian error linear units (gelus).
- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.
- Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.
- Jaccard, P. (1901). Etude de la distribution florale dans une portion des alpes et du jura. *Bulletin de la Societe Vaudoise des Sciences Naturelles*, 37:547–579.
- Jacovi, A., Sar Shalom, O., and Goldberg, Y. (2018). Understanding convolutional neural networks for text classification. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 56–65, Brussels, Belgium. Association for Computational Linguistics.
- Jean, S., Cho, K., Memisevic, R., and Bengio, Y. (2015). On using very large target vocabulary for neural machine translation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1–10, Beijing, China. Association for Computational Linguistics.
- Jurafsky, D. and Martin, J. H. (2019). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition.
- Kalchbrenner, N., Espeholt, L., Simonyan, K., van den Oord, A., Graves, A., and Kavukcuoglu, K. (2016). Neural machine translation in linear time.
- Khomenko, V., Shyshkov, O., Radyvonenko, O., and Bokhan, K. (2017). Accelerating recurrent neural network training using sequence bucketing and multi-gpu data parallelization.
- Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.
- Le Cun, Y., Jackel, L. D., Boser, B., Denker, J. S., Graf, H. P., Guyon, I., Henderson, D., Howard, R. E., and Hubbard, W. (1989). Handwritten digit recognition: applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46.
- Lee, J., Cho, K., and Hofmann, T. (2017). Fully character-level neural machine translation without explicit segmentation. *Transactions of the Association for Computational Linguistics*, 5:365–378.

BIBLIOGRAPHY

- Lin, C.-Y. (2004). ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.
- Liu, P., Qiu, X., and Huang, X. (2016). Recurrent neural network for text classification with multi-task learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI’16*, pages 2873–2879. AAAI Press.
- Luong, M.-T., Pham, H., and Manning, C. D. (2015a). Effective approaches to attention-based neural machine translation.
- Luong, M.-T., Sutskever, I., Le, Q. V., Vinyals, O., and Zaremba, W. (2015b). Addressing the rare word problem in neural machine translation.
- Maas, A. L. (2013). Rectifier nonlinearities improve neural network acoustic models.
- Marzal, A. and Vidal, E. (1993). Computation of normalized edit distance and applications. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(9):926–932.
- McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.
- Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space.
- Ng, A. (2017). Improving deep neural networks: Hyperparameter tuning, regularization and optimization.
- Pang, B., Lee, L., and Vaithyanathan, S. (2002). Thumbs up? sentiment classification using machine learning techniques. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP 2002)*, pages 79–86. Association for Computational Linguistics.
- Papineni, K., Roukos, S., Ward, T., and jing Zhu, W. (2002). Bleu: a method for automatic evaluation of machine translation. pages 311–318.
- Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *In EMNLP*.
- Rikters, M. and Fishel, M. (2017). Confidence through attention.
- Rong, X. (2014). word2vec parameter learning explained.
- Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386.
- Salimans, T. and Kingma, D. P. (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks.
- See, A., Liu, P. J., and Manning, C. D. (2017). Get to the point: Summarization with pointer-generator networks.
- Sennrich, R., Haddow, B., and Birch, A. (2015). Neural machine translation of rare words with subword units.
- Sutskever, I., Vinyals, O., and Le V, Q. (2014). Sequence to sequence learning with neural networks.
- Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2015). Rethinking the inception architecture for computer vision.
- Tieleman, H. (2012). rmsprop: Divide the gradient by a running average of its recent magnitude.
- Tu, Z., Lu, Z., Liu, Y., Liu, X., and Li, H. (2016). Modeling coverage for neural machine translation.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need.

BIBLIOGRAPHY

- Wang, X., Jiang, W., and Luo, Z. (2016). Combination of convolutional and recurrent neural network for sentiment analysis of short texts. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 2428–2437, Osaka, Japan. The COLING 2016 Organizing Committee.
- Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Łukasz Kaiser, Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016). Google’s neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144.