

Deep Inverse Cooking

VM 02
MASTER OF SCIENCE IN ENGINEERING

presented by
MARC BRAVIN

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

June 30, 2020

Advisor Prof. Dr. Marc Pouly
Lucerne University of Applied Sciences and Arts, 6343 Rotkreuz, Switzerland
marc.pouly@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.

Abstract

Medical images are widely used in hospitals for the diagnosis and treatment of many diseases, such as skin cancer or diabetic retinopathy. Machine learning algorithms have recently been shown to outperform human doctors in a broad variety of diagnosis tasks. A diagnosis is often posed as a semantic segmentation problem where models are trained to classify each pixel of an image or as a multi-label classification task where the output is a set of tags. However, both types of outputs are hard to interpret due to the lack of reasoning about how the decisions were achieved. In contrast, a diagnosis made by a medical doctor is different. When a family doctor refers a patient to a specialist, he will expect a medical report in which the specialist explains her diagnosis. Likewise, the output of a neural network would be more useful if augmented by a medical report written in a natural language.

Recently, there has been much progress in the development of image-to-text models that the task of automatically generating medical reports can now be considered feasible. However, such models require a large amount of paired data, i.e. images paired with medical reports. To the author's best knowledge, there is no publicly available dataset of such paired data. In order to experiment with image-to-text models, domains were switched from medicine to cooking, where such data is prolific. A dataset consisting of 0.9M recipes and 1.3M images was acquired through crawling five different cooking platforms. Since the majority of the recipes originate from community cooking websites, an extensive data cleaning pipeline had to be implemented. This allowed the number of unique ingredients to be reduced from 1M to 1.3k at the cost of dropping some recipes.

Using this dataset, a multi-task neural network model was implemented, trained and evaluated. It generates a list of ingredients (cf. medical features), a title and cooking instructions (cf. medical report) based on an image of a dish. The model consists of a VGG-16 encoder to extract image features. Given these features, a transformer-based decoder generates a list of ingredients. Finally, an additional transformer decoder generates the recipe title as well as the cooking instructions by processing the image and ingredients features simultaneously. Evaluation on unseen test data showed that the model achieves an F_1 score of 38.62% for the ingredients prediction, a $BLEU_1$ score of 7.17% for generating the title and a $BLEU_4$ score of 6.15% for the instructions text generation task. Comparing the architecture of the inverse cooking model to medical image captioning systems from the literature shows several similarities. Therefore, it is expected that the proposed model can be adapted and extended for generating medical reports in the future.

Contents

1	Introduction	1
1.1	Research Questions and Expected Results	1
1.2	Organization of this Report	2
1.2.1	Notation	2
1.3	Acknowledgements	2
2	Related Work	3
2.1	Artificial Neural Networks	3
2.1.1	Perceptron	3
2.1.2	Activation Functions	5
2.1.3	Cost Functions	9
2.1.4	Gradient Descent Algorithm	9
2.1.5	Stabilizing Gradients	11
2.1.6	Transfer Learning	12
2.2	Convolutional Neural Networks	13
2.2.1	Convolutional Layers	13
2.2.2	Pooling Layers	15
2.2.3	Network Architectures	15
2.3	Autoregressive Models	17
2.3.1	Recurrent Neural Networks	17
2.3.2	Attention Mechanism	20
2.3.3	Transformer Model	20
2.4	Multi-label Image Classification	21
2.4.1	Classical Approaches	21
2.4.2	Autoregressive Approaches	21
2.4.3	Metrics	22
2.5	Word Embeddings	24
2.5.1	Term Frequency-Inverse Document Frequency	24
2.5.2	Word2Vec	24
2.5.3	GloVe	25
2.5.4	FastText	26
2.5.5	Flair	26
2.6	Language Models	27
2.6.1	BERT Language Model	27
2.7	Image-to-Text Models	28
2.7.1	Traditional Retrieval Approaches	28
2.7.2	Encoder-Decoder Models	29
2.7.3	Encoder-Decoder Models with Attention	30
2.7.4	Dense Captioning	33
2.7.5	Hierarchical Approach	33
2.7.6	CNN+CNN Models	34
2.7.7	Approaches using Transformer Networks	35
2.7.8	Decoding Strategies	38
2.7.9	Metrics	40
2.7.10	Training	45
2.8	Inverse Cooking	47
2.8.1	Recipe Retrieval	47
2.8.2	Recipe Generation	49
2.8.3	Metrics	50

3	Setup and Models	51
3.1	Data Quality Assessment	51
3.1.1	Data Cleaning	51
3.1.2	Data Analysis	56
3.2	Preprocessing	60
3.2.1	Dataset Splitting	60
3.2.2	Input Pipeline	60
3.3	Metrics	63
3.3.1	Recipe Title	63
3.3.2	Ingredients	63
3.3.3	Instructions	63
3.4	Baseline Model	65
3.5	Inverse Cooking Model	66
3.5.1	Image Encoder	66
3.5.2	Ingredients Decoder	67
3.5.3	Instructions Decoder	69
3.5.4	Training	69
3.5.5	Inference	70
4	Results	71
4.1	Model Comparison	71
4.1.1	Metrics	71
4.1.2	Number of Trainable Parameters	72
4.1.3	Model Selection	73
4.1.4	Shuffling Ingredients	73
4.1.5	Beam Search	73
4.2	Final Evaluation	73
4.2.1	Ingredients	74
4.2.2	Qualitative Evaluation	75
5	Discussion	79
5.1	Outlook	79
5.2	Adaption to Medical Domain	80
5.3	Conclusion	81
A	Appendix	82
A.1	Demonstration App	82
A.2	Code	83
A.2.1	Jupyter Notebooks	83
A.2.2	Recipe Model	83

Chapter 1

Introduction

This chapter provides an introduction to the topic of image-to-text models and explains the motivation behind this project.

1.1 Research Questions and Expected Results

Medical images are widely used in hospitals for the diagnosis and treatment of many diseases, such as skin cancer or diabetic retinopathy. Machine learning algorithms have recently been shown to outperform human doctors in a broad variety of important diagnosis tasks. In general, machine learning based diagnosis algorithms for medical images often have their origins in computer vision. They are either posed as semantic segmentation tasks where a model is trained to classify each pixel of an image or multi-label classification where the output is a set of tags describing the diagnosis. This surge in accuracy came at the cost of increased model complexity. Whereas humans are able to argue and explain how they reach a decision, deep learning models appear as black box mechanisms. We have only a very limited understanding of what grounds such models come up with their decisions. This is a serious issue especially in medicine, where a misdiagnosis or error can result in the wrong treatment and therefore have severe repercussions for the affected patients. In order to be able to use deep learning models in everyday clinical life, we need to make the decisions of a deep learning model comprehensible and transparent to physicians, patients and other stakeholders such as insurance companies.

Many different techniques and frameworks for explainable deep learning have been proposed. Most of them are focussing on visually explaining the network's output with respect to the input image such as highlighting the most influential image regions. Explainability in the medical domain is different though. When a family doctor refers a patient to a specialist, he will expect a medical report in which the specialist explains her diagnose. When a family doctor refers a patient to a specialist, he will expect a medical report in which the specialist explains her diagnose. Likewise, a neural network analysing an image should detail its diagnose with a medical report in natural (domain) language targeted to a human doctor. Recently, there has been much progress in the development of image-to-text models that combine computer vision with natural language processing. Therefore, the task of automatically generating medical reports can now be considered feasible. However, such models require a large amount of paired data, i.e. images paired with medical reports. To the author's best knowledge, no publicly available dataset of such paired data exists, and the University Hospital Basel has started acquisition of paired data only recently. Moreover, privacy concerns weigh even more as medical reports contain much more personal data than images.

In order to experiment with image-to-text models, domains were switched from medicine to cooking, where such data is abundant. Based on a photography of food or dishes, a neural network model should come up with a list of ingredients (cf. medical features), a recipe title and cooking instructions (cf. medical report). To train such models, data should be acquired through web crawling and analysed by means of a data quality assessment. Furthermore, a state-of-the-art research on image-to-text models should be conducted.

1.2 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 how the implemented models can be applied in the medical domain.

1.2.1 Notation

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

t Discrete time step

\mathbb{R} Set of real numbers

\cdot Multiplication

\odot Element-wise multiplication

\star Convolution operation

θ Parameters of the machine learning model

\mathbf{y} Vector of the ground truth values

$\hat{\mathbf{y}}$ Vector of the model predictions

\mathcal{L} Cost or loss function

σ Sigmoid activation 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

$[\mathbf{A}; \mathbf{B}]$ Concatenation of the vectors \mathbf{A} and \mathbf{B}

1.3 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.

Chapter 2

Related Work

This chapter introduces the concepts used throughout this report. It starts with the basics of neural networks and explains how convolutional and recurrent neural networks work. Then it introduces several tasks in the field of computer vision, such as multi-label image classification. Moreover, the main concepts of natural language processing are reviewed. Finally, the topics of computer vision and natural language processing are combined by introducing image-to-text models. The chapter concludes by describing the possibilities of machine learning in the domain of cooking.

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 neurons in the brains of multicellular organisms such as humans. They describe the brain as a net of neurons where the neurons send an impulse when 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.

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

Figure 2.1 shows a schematic representation of an 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.

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.

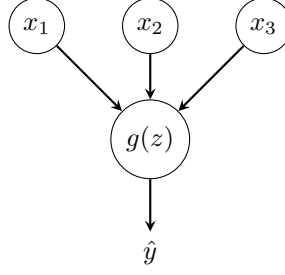


Figure 2.1: A graphical illustration of an MP neuron proposed by McCulloch and Pitts (1943)

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

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

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 .

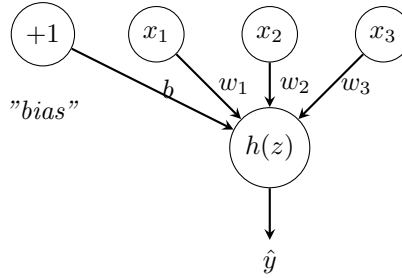


Figure 2.2: A graphical illustration of a perceptron proposed by Rosenblatt (1958)

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

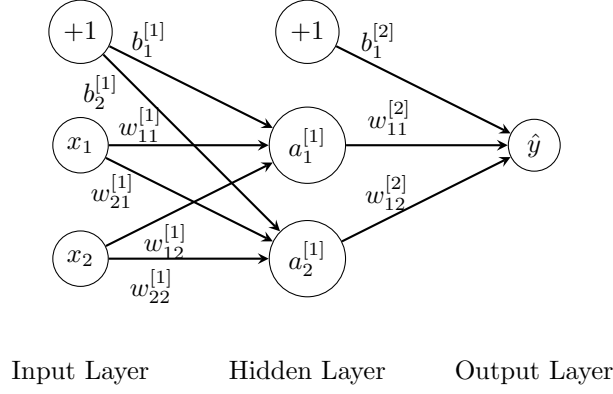


Figure 2.3: An example of an MLP with one hidden layer and two inputs

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 binary classification problems. 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 computed as follows:

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

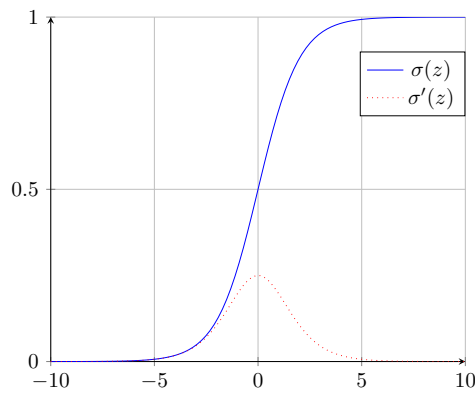


Figure 2.4: The sigmoid activation function

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.

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

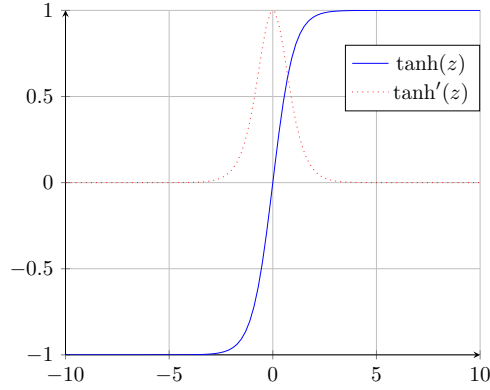


Figure 2.5: The tanh activation function

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 states 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.

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

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

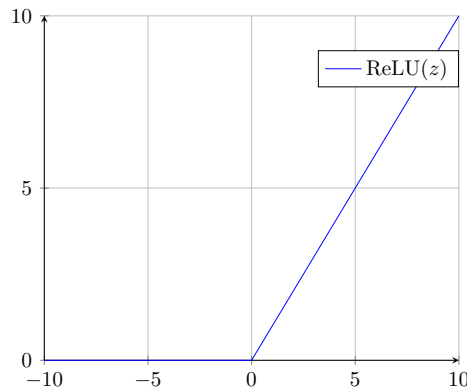


Figure 2.6: The ReLU activation function

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 $\alpha = 0.1$.

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

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

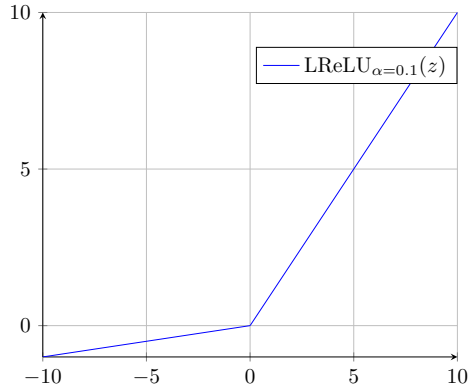


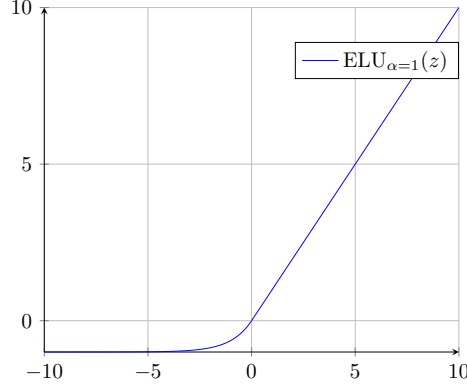
Figure 2.7: The leaky ReLU activation function with $\alpha = 0.1$

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 2.7.7). 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 specified in equation 2.14 and can be approximated using a tanh or sigmoid σ function. Figure 2.9 shows a plot of the function.

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

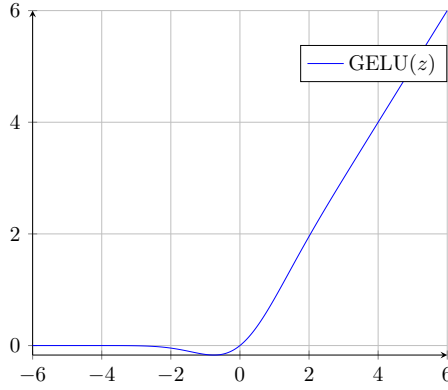


Figure 2.9: The GELU activation function

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 \mathcal{L} needs to be defined.

Mean Squared Error Cost Function

The mean squared error (MSE) cost function \mathcal{L}_{MSE} is typically used for regression problems. It calculates the sum of the squared differences 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.

$$\mathcal{L}_{\text{MSE}}(\theta) = \frac{1}{2m} \sum_{i=1}^m (\hat{y}^{[i]} - y^{[i]})^2 \quad (2.16)$$

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.

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

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

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

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 \mathcal{L} and all activation functions of the neural network need to be differentiable. Algorithm 1 shows how the gradient descent works. The idea behind the algorithm is that the gradient $\Delta\mathcal{L}(\theta) = \frac{\partial\mathcal{L}}{\partial\theta}$ of the cost function \mathcal{L} 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. until changes in the parameter fall below a certain threshold.

Algorithm 1 Gradient descent algorithm

initialize parameters θ_t

repeat

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

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 by Sutskever et al. (2013) 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  $\theta_t, \mathbf{m}_t$ 
repeat
     $\mathbf{m}_{t+1} \leftarrow \beta_1 \cdot \mathbf{m}_t + \alpha \cdot \Delta \mathcal{L}(\theta_t)$ 
     $\theta_{t+1} \leftarrow \theta_t - \mathbf{m}_{t+1}$ 
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  $\theta_t, \mathbf{s}_t$ 
repeat
     $\mathbf{s}_{t+1} \leftarrow \beta_2 \cdot \mathbf{s}_t + (1 - \beta_2) \cdot \Delta \mathcal{L}(\theta_t) \odot \Delta \mathcal{L}(\theta_t)$ 
     $\theta_{t+1} \leftarrow \theta_t - \frac{\alpha}{\sqrt{\mathbf{s}_{t+1} + \epsilon}} \odot \Delta J(\theta_t)$ 
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 θ .

Algorithm 4 Adam algorithm

```

initialize parameters  $\theta_t, \mathbf{m}_t, \mathbf{s}_t$ 
repeat
     $\mathbf{m}_{t+1} \leftarrow \beta_1 \cdot \mathbf{m}_t + (1 - \beta_1) \cdot \Delta \mathcal{L}(\theta_t)$  // Update biased first moment estimate
     $\mathbf{s}_{t+1} \leftarrow \beta_2 \cdot \mathbf{s}_t + (1 - \beta_2) \cdot \Delta \mathcal{L}(\theta_t) \odot \Delta \mathcal{L}(\theta_t)$  // Update biased second moment estimate
     $\tilde{\mathbf{m}}_{t+1} \leftarrow \frac{\mathbf{m}_{t+1}}{1 - \beta_1}$  // Compute bias-corrected first moment estimate
     $\tilde{\mathbf{s}}_{t+1} \leftarrow \frac{\mathbf{s}_{t+1}}{1 - \beta_2}$  // Compute bias-corrected second moment estimate
     $\theta_{t+1} \leftarrow \theta_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{s}}_{t+1} + \epsilon}} \odot \Delta \mathcal{L}(\theta_t)$  // 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. The term 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 2.3) 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-batches, they normalize the weights of the layers. To do so, the authors propose to parametrize each weight vector \mathbf{w} in terms of a vector \mathbf{v} and a scalar g according to equation 2.19 where $\|\mathbf{v}\|_2$ 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.

$$\mathbf{w} = \frac{g}{\|\mathbf{v}\|_2} \mathbf{v} \quad (2.19)$$

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.

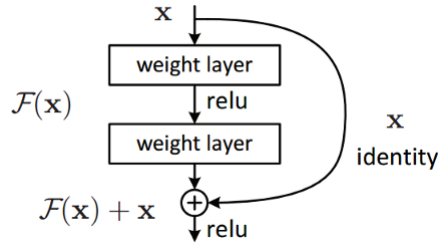


Figure 2.10: A visualization of a residual block by He et al. (2015)

2.1.6 Transfer Learning

Torrey and Shavlik (2009) describe transfer learning as the improvement of learning in a new task through the transfer of knowledge from a related task that has already been learned. In the context of neural networks, transfer learning is a method where a network is trained on a base dataset and then later repurposed on a second network to be trained on a target dataset and task.

Yosinski et al. (2014) explain that when training neural networks for image classification, earlier layers tend to learn standard features such as Gabor filters or color blobs. This phenomenon occurs not only for different datasets but also for different training objectives. However, the final layers of networks depend greatly on the chosen dataset and task. Transfer learning can be applied by using the first n layers of a pre-trained neural network as feature extraction. Depending on the dataset size of the target task, it makes sense to freeze some of the layers to prevent the model from overfitting.

2.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of neural networks especially suited for processing grid-like data such as images. They were proposed by Le Cun et al. (1989) for handwritten digit recognition. Although they were introduced in the late 1980s, they remained unpopular until Krizhevsky et al. (2012a) used them to win the ImageNet challenge. A CNN consists of several layers that are stacked onto each other. The following sections describe the most common layers that are used with CNNs as well as popular CNN architectures.

2.2.1 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 matrix $\mathbf{F} \in \mathbb{R}^{w_f \times h_f}$ is moved across an input $\mathbf{I} \in \mathbb{R}^{w_i \times h_i}$ and multiplied with the current part of the input. The products are then aggregated and result in a single scalar for each output position. Each filter consists of a bias b that is added to the result. Equation 2.20 shows how the output \mathbf{O} at positions i and j can be calculated where \star denotes the convolution operation. The filters and inputs are often squares, i.e. $w_i = h_i$ and $w_f = h_f$.

$$O_{i,j} = (\mathbf{F} \star \mathbf{I})(i,j) = \sum_{m=1}^{w_f} \sum_{n=1}^{h_f} F_{m,n} \cdot I_{i+m,j+n} + b \quad (2.20)$$

How many pixels a filter moves at each step is defined by means of the stride size s . 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 p , that specifies the size of a zeroed frame added around the input. Equation 2.21 shows how the output dimensions $w_o \times h_o$ are calculated if a convolution layer with a filter size $w_f \times h_f$, a padding size of p and a stride of s is applied to an input of size $w_i \times h_i$.

$$\begin{aligned} w_o &= \left\lfloor \frac{w_i - w_f + 2p}{s} \right\rfloor + 1 \\ h_o &= \left\lfloor \frac{h_i - h_f + 2p}{s} \right\rfloor + 1 \end{aligned} \quad (2.21)$$

In general, it can be distinguished between two padding strategies:

Same The goal of this strategy is that the output after applying the convolutional layer has the same shapes as the inputs. Therefore, the padding p is selected so that $w_o = (w_i + 2p - k) \cdot s$ and $h_o = (h_i + 2p - k) \cdot s$ are fulfilled.

Valid With the valid strategy, the padding is set to zero. Therefore, the output might not have the same shape as the input.

Figure 2.11 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 .

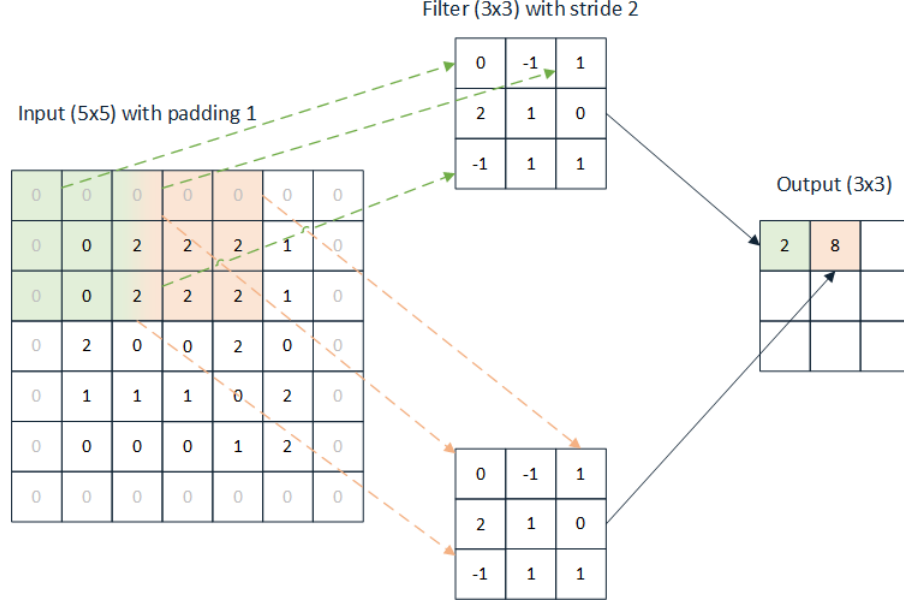


Figure 2.11: A CNN layer with size 3×3 , stride = 2 and padding = 1 applied to an input of dimension 5×5

Traditional feed-forward networks have a matrix of parameters, connecting each input unit with each output unit, which is referred to as dense connectivity. Figure 2.12 shows that the highlighted input x_3 affects all output units.

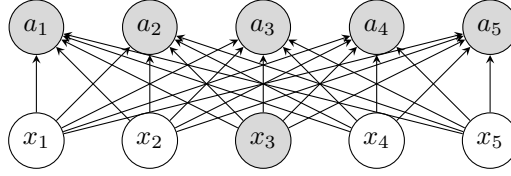


Figure 2.12: Illustration of the dense connectivity of feed-forward networks

Goodfellow et al. (2016) describe that convolutional networks have sparse interactions by making the kernel smaller than the input. This means that fewer parameters need to be stored which not only reduces the memory requirements but also improves its efficiency. Computing the output requires fewer operations. Figure 2.13 depicts this sparse connectivity where now the input x_3 only affects three output units if a kernel of size 3 is used.

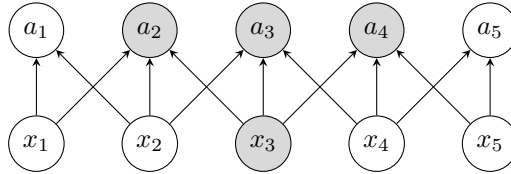


Figure 2.13: Illustration of the sparse connectivity of convolutional networks

2.2.2 Pooling Layers

A pooling layer reduces the spatial size of the activation map. It 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 2.14 shows an example of a max-pooling and an average-pooling operation 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.

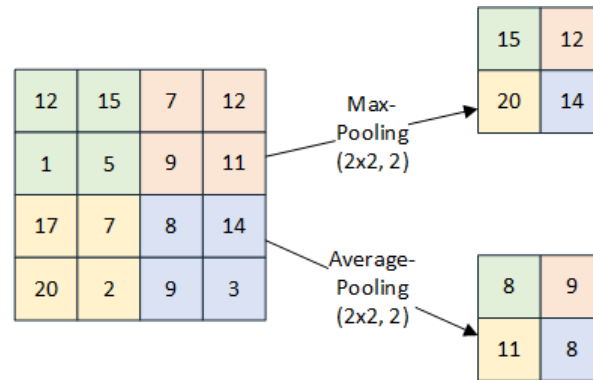


Figure 2.14: An example of max-pooling and average-pooling with a pooling size 2×2 and stride 2

2.2.3 Network Architectures

The research of CNNs is characterized by a tendency to increase the number of hidden layers, i.e. to construct deeper networks. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is an annual image classification challenge (Russakovsky et al., 2014) that contributed significantly to the development of state-of-the-art CNN architectures. Participants are required to evaluate their models on the ImageNet image classification dataset that contains over a million images with 1'000 different classes. The following sections explain the most important architectures that were proposed in order to solve image classification and to win the ILSVRC challenge.

LeNet LeCun et al. (1998) proposed LeNet, a neural network that was trained for handwritten digit recognition. It was one of the first successful applications of CNNs. The model consists of several convolutional and pooling layers for feature extraction, followed by a fully-connected layer used for classification.

AlexNet Krizhevsky et al. (2012b) won the ILSVRC-2012 challenge with their AlexNet model architecture. It was the first time that a CNN-based model won the challenge since it had started in 2010. The model consists of 5 convolutional, 3 max-pooling and 3 fully-connected layers.

VGG Simonyan and Zisserman (2014) achieved the second place of the ILSVRC-2014 challenge by increasing the number of layers significantly. Their architecture is based on the AlexNet, however, they started the trend of using smaller filters with only size 3×3 and stride 1. There are variants of the network with 16 and 19 layers each, which are referred as VGG-16 and VGG-19 respectively.

GoogLeNet The GoogLeNet model was introduced by Szegedy et al. (2014) with a total of 22 layers. To be able to stack so many layers without introducing the problem of vanishing gradients, they used two additional auxiliary outputs that are connected to the feature extractors. During training, gradients are injected at these additional outputs to diminish the effect of vanishing gradients. One drawback of using CNNs is that they introduce the filter size as an additional parameter that needs to be tuned. Instead of using only 1 filter size per layer, the authors introduced a new layer which they called inception module. It is composed of several convolutional filters in parallel with different filter sizes as depicted in figure 2.15. To reduce the depth while preserving the spatial dimensions, they use 1×1 convolutional layers. This way, despite its increased number of layers, the model requires about 12 times fewer parameters than VGG. By introducing these new concepts, GoogLeNet won the ILSVRC-2014 competition. Later, the

authors refined this approach by stacking 48 layers and named the model Inception-V3 (Szegedy et al., 2016).

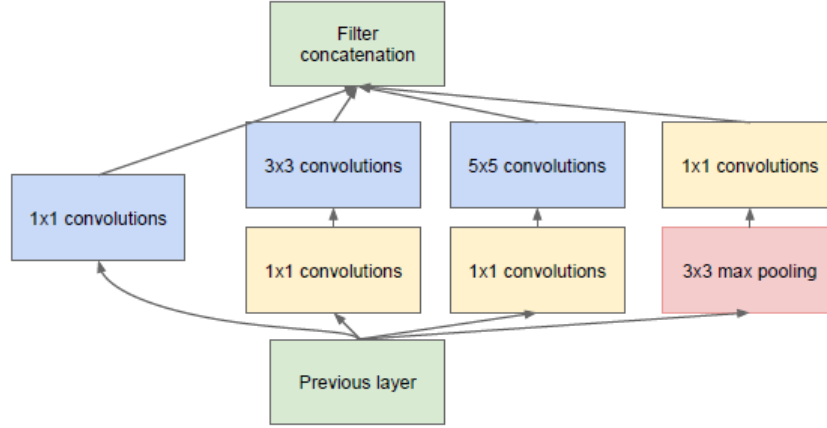


Figure 2.15: An inception module introduced with the GoogLeNet by Szegedy et al. (2014)

ResNet He et al. (2015) published ResNet, which uses residual blocks (see section 2.1.5) that allowed them to build an even deeper neural network. Thanks to these residual blocks, they won the ILSVRC-2015 competition. There are several variants of the ResNet architecture, for example the ResNet-50 and ResNet-101, which consist of 50 and 101 stacked layers respectively.

ResNeXt Xie et al. (2016) secured the 2nd place of the ILSVRC-2016 competition with their proposed architecture. It is referred to as ResNeXt because it can be viewed as an extension of ResNet. They introduced a new hyperparameter *cardinality* C which defines the number of parallel blocks. Figure 2.16 shows a ResNeXt model with cardinality 1 on the left, which is equal to the ResNet architecture. On the right is a model with a cardinality of 32.

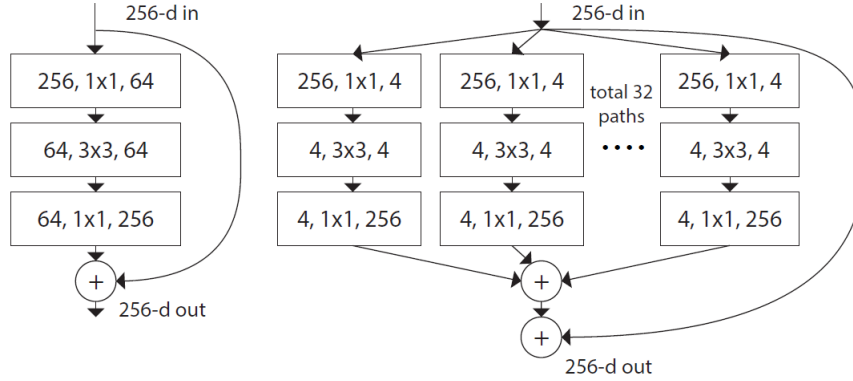


Figure 2.16: Visualization of the ResNet (He et al., 2015) and the ResNeXt (Xie et al., 2016) architecture

2.3 Autoregressive Models

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 do not use the results from the previous inputs to make its decisions. Autoregressive models are designed to process input data of variable length and have a sort of memory that allows them to access previously generated output. There are two different types of autoregressive models that are commonly used in the deep learning community: recurrent neural networks and transformer models.

2.3.1 Recurrent Neural Networks

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 x_t . Equation 2.22 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. Figure 2.17 illustrates an RNN that processes an input sequence.

$$\mathbf{h}_t = \begin{cases} 0 & t = 0 \\ f(\mathbf{h}_{t-1}, x_t) & \text{else} \end{cases} \quad (2.22)$$

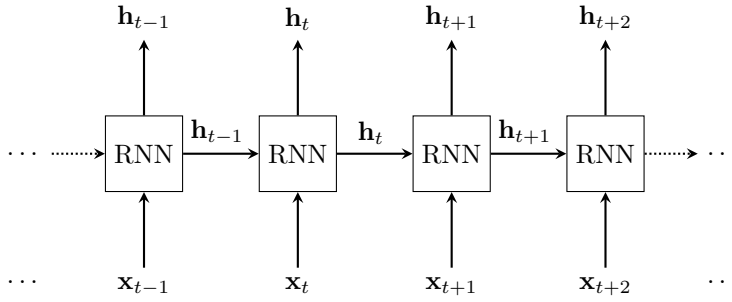


Figure 2.17: Visualization of an RNN unfolded over time

In a standard recurrent cell the activation function f is usually a tanh function. Figure 2.18 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 2.23 where \mathbf{W}_x , \mathbf{W}_h and \mathbf{b}_h are learnable parameters.

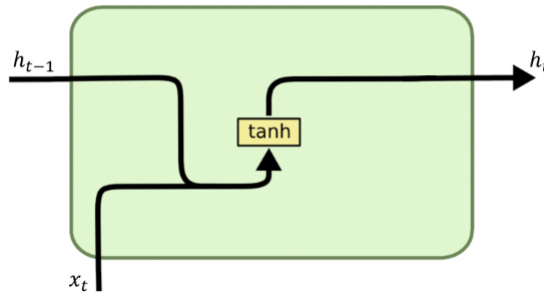


Figure 2.18: Visualization of a simple RNN cell by Colah (2015)

$$\mathbf{h}_t = \tanh(\mathbf{W}_x \cdot \mathbf{x}_t + \mathbf{W}_h \cdot \mathbf{h}_{t-1} + \mathbf{b}_h) \quad (2.23)$$

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 2.24. This learned distribution can be used to sample a new sequence by iteratively sampling a symbol at each time step t .

$$p(\mathbf{x}) = \prod_{t=1}^T p(x_t | x_{t-1}, \dots, x_1) \quad (2.24)$$

RNNs are used whenever the 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 since the multiplicative structure can lead to vanishing or exploding gradients.

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 parts of the previous hidden state are used to compute the new content. Figure 2.19 visualizes a GRU cell and algorithm 7 shows how to compute the next hidden state \mathbf{h}_t .

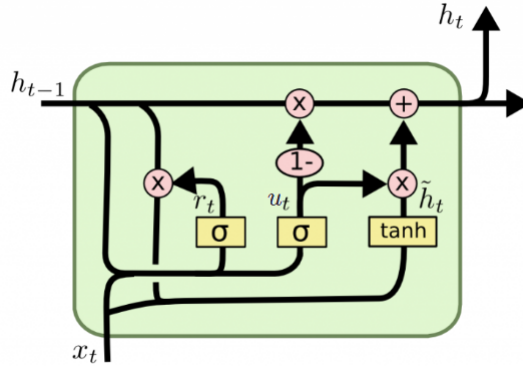


Figure 2.19: Visualization of a GRU cell by Colah (2015)

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

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

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

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

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

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 2.20 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 time step t and \mathbf{C}_t the cell state at time step t .

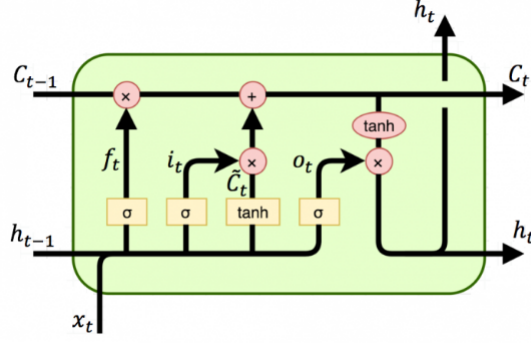


Figure 2.20: Visualization of a LSTM cell by Colah (2015)

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

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

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

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

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

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

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

Bidirectional Recurrent Neural Networks

Schuster and Paliwal (1997) extend classical RNNs to bidirectional RNNs that consists of a forward and a backward RNN. The forward RNN reads the sequence in its ordered way, whereas the backward RNN reads it in reverse order. The output of such a bidirectional RNN (biRNN) is a forward $\vec{\mathbf{h}}_t$ and a backward hidden state $\overleftarrow{\mathbf{h}}_t$ which are then combined to \mathbf{h}_t . This can provide an additional context to the network and result in better performance. However, such networks can only be used for problems where all time steps of the input sequence are available. Figure 2.21 visualizes the architecture of a bidirectional RNN.

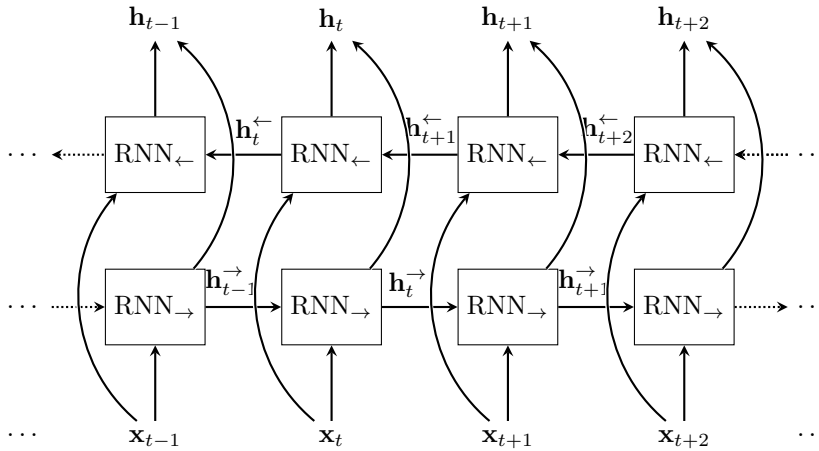


Figure 2.21: Visualization of a bidirectional RNN unfolded over time

2.3.2 Attention Mechanism

RNNs have been successfully applied to many natural language processing (NLP) and computer vision tasks such as machine translation or image captioning. Although LSTM and GRU cells are extensions that were specifically designed to handle long sequences, they have still the issue that the history of the previously processed data is stored in a single vector \mathbf{h}_t . Recent work has proposed the attention mechanism that allows the network to focus on portions of the input to predict an output. Given L inputs $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_L)$ and the hidden state \mathbf{h}_t of an RNN at time step t , the attention mechanism computes a score $e_{t,i}$ that measures how good the hidden state \mathbf{h}_t at time step t matches with the input \mathbf{x}_i at position i . It is computed as follows:

$$e_{t,i} = \text{score}(\mathbf{h}_t, \mathbf{x}_i) = \begin{cases} \mathbf{W}_a^\top \tanh(\mathbf{W}_b[\mathbf{x}_i; \mathbf{h}_t]) & \text{Additive Attention} \\ \mathbf{h}_t^\top \mathbf{W}_a \mathbf{x}_i & \text{Multiplicative Attention} \\ \frac{\mathbf{h}_t^\top \mathbf{x}_i}{\sqrt{L}} & \text{Scaled Dot-Product Attention} \end{cases} \quad (2.25)$$

Many different attention scores were proposed. The most common ones are the additive attention (Bahdanau et al., 2014), multiplicative attention (Luong et al., 2015) and scaled dot-product attention (Vaswani et al., 2017). Both \mathbf{W}_a and \mathbf{W}_b are learnable parameters and are jointly trained with the model. These scores are then passed to a softmax function to obtain the attention weights α :

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^L \exp(e_{t,k})} \quad (2.26)$$

Finally, a context vector \mathbf{c}_t is obtained by computing a weighted sum of the input using the attention weights as given in equation 2.27. This vector includes information from previous time steps and can be used for further processing instead of the hidden state \mathbf{h}_t .

$$\mathbf{c}_t = \sum_{i=1}^L \alpha_{t,i} \cdot \mathbf{x}_i \quad (2.27)$$

While the input \mathbf{x}_i is often used to compute both the attention weights and the context vector, it is possible to have separate representations for each operations. Regarding the terminology for attention mechanisms, the vector \mathbf{x}_i in equation 2.25 is referred to as *key*, whereas in equation 2.27 as *value*. The hidden state \mathbf{h}_t is commonly called *query*. How the attention mechanism can be used for image-to-text models is explained in depth in section 2.7.3.

2.3.3 Transformer Model

RNNs are slow to train due to their autoregressive nature in training as well as during inference. To compute the output at time step t , all previous time steps need to be processed first. Vaswani et al. (2017) proposed the *transformer*, an autoregressive model that is purely based on the attention mechanism. To allow for parallelization, the model is trained with teacher forcing (Williams and Zipser, 1989) which removes the dependency to previous outputs and thus allows faster training times. To account for the word orders, each feature is positionally encoded. Section 2.7.7 describes the transformer in more detail in the context of image-to-text models.

2.4 Multi-label Image Classification

With the introduction of the ILSVRC challenge and huge datasets such as ImageNet, many architectures have been proposed to solve the image classification task. However, real-world images generally contain multiple labels that correspond to different objects, scenes or actions. Tasks like image segmentation or object detection account for this increased scene complexity. However, they require detailed annotations which are expensive to obtain. Multi-label image classification assigns multiple labels to an image and is often used in social media platforms, for example for tagging uploaded images. Image labels are usually unordered, i.e. permuting them does not change their meaning. Therefore, multi-label image classification can be seen as an image-to-set prediction problem where a set $\mathbf{y} = \{y_1, \dots, y_T\}$ of length T is generated based on an image \mathbf{I} . Each element in the set may be selected at most once from a dictionary $\mathcal{D} = \{d_1, \dots, d_K\}$ of size K .

Despite the great success of CNNs for image classification, they can not be trivially extended to cope with multi-label image classification tasks. Typically, the output of CNNs have a fixed-sized shape, such as vectors for image classification or matrices for image segmentation.

2.4.1 Classical Approaches

Nam et al. (2014) describe that a naive approach for multi-label image classification would be to treat each label independently and train multiple binary classifiers. Another method is known as problem transformation (Read, 2010), which converts the multi-label problem into multiple single-label classification problems using classifier chains. Other approaches handle multiple labels by sampling different regions from an image and then classify each with a single-label image classifier. A more sophisticated approach has been proposed by Wei et al. (2014) where they first generate multiple object proposals and then classify each.

Pineda et al. (2019) describe an approach where they represent the output as a binary vector $\mathbf{y} = (y_1, \dots, y_K)$, $y_i \in \{0, 1\}$, where $y_i = 1$ if label d_i is selected and $y_i = 0$ otherwise. Then they use a pre-trained CNN to extract image features \mathbf{X} which they flatten and feed into a fully-connected layer followed by a sigmoid activation function to obtain the predictions $\hat{\mathbf{y}} = (\hat{y}_1, \dots, \hat{y}_K)$. They optimize the model using a binary cross entropy loss.

These aforementioned approaches have the drawback that they do not model the dependency between multiple labels. For example, in an ingredient detection system *salt* and *pepper* usually appear together, whereas *pepper* and *sugar* almost never co-occur. One way to capture label dependencies is to use label powersets (Tsoumakas and Vlahavas, 2007), considering all possible label combinations. However, computing powersets makes it intractable for problems with a large number of classes. Read et al. (2011) make use of label dependencies by training a chain of binary classifiers, where each classifier predicts whether the current label exists given the image features and the labels already predicted.

2.4.2 Autoregressive Approaches

To address the scalability issue, Wang et al. (2016) applied an RNN to decompose the joint distribution into conditionals. They represent the output as a binary matrix $\mathbf{y} \in \{0, 1\}^{T \times K}$ where the value of $y_{t,i}$ is set to 1 if label d_i is selected at the t -th position and 0 otherwise. To generate the output matrix $\hat{\mathbf{y}}$, they use a CNN to extract features from the image and then condition the RNN on these features and the previously predicted labels. To train the model, they optimize the cross entropy loss at each time step. There is a main disadvantage of using autoregressive models for multi-label classification tasks. Since predicting the next element of the set requires the previous output, they introduce an intrinsic label ordering during training and therefore do not respect the set property. There are several strategies to predefine label orders, for example by ordering them based on their frequency or simply alphabetically.

Chen et al. (2017b) extended the work from Wang et al. (2016) by applying the attention mechanism during the generation of the output. To solve the issues of pre-determining the label order, they optimize for the most likely ground truth label at each time step. Pineda et al. (2019) set activations of previously selected labels to $-\infty$ to ensure that labels are selected without repetition. To ignore the order in which labels are predicted, they aggregate the outputs across different time steps by means of a max-pooling operation. They then minimize the binary cross entropy between the pooled predicted labels and the ground truth.

2.4.3 Metrics

For traditional single-label image classification, accuracy is the most common evaluation metric. Despite its popularity, He and Garcia (2009) describe that the accuracy score has to be taken with care when the classes are imbalanced. With an imbalanced dataset, high accuracy is achieved if the classifier simply predicts the majority class for all examples. Therefore, other metrics such as precision or recall score are preferred when working with imbalanced data. Sorower (2010) explains that in multi-label classification, the prediction can be either *fully correct*, *partially correct* or *fully incorrect*. None of the existing metrics for single-label classification capture such notion in their original form which makes evaluation of multi-label classification more challenging.

Multi-label classification metrics can be distinguished between example-based and label-based methods. Example-based methods compute the metrics per example and then average them. Godbole and Sarawagi (2004) describe the following metrics, where N denotes the number of samples in the dataset, \mathbf{y} a set of ground truth values, and $\hat{\mathbf{y}}$ the predictions.

IoU The intersection over union (IoU) measures the proportion of the number of predicted correct labels to the total number of labels for that instance. It is averaged over the number of samples n and computed as follows:

$$\text{IoU} = \frac{1}{N} \sum_{i=1}^N \frac{|y_i \cap \hat{y}_i|}{|y_i \cup \hat{y}_i|} \quad (2.28)$$

Recall The recall score measures the completeness, i.e. how many examples were predicted correctly. It is computed as the proportion of predicted correct labels to the total number of actual labels:

$$\text{Recall} = \frac{1}{N} \sum_{i=1}^N \frac{|y_i \cap \hat{y}_i|}{|y_i|} \quad (2.29)$$

Precision The precision score measures the exactness, i.e. how many of the examples were actually predicted correctly. It is computed as the proportion of predicted correct labels to the total number of predicted labels:

$$\text{Precision} = \frac{1}{N} \sum_{i=1}^N \frac{|y_i \cap \hat{y}_i|}{|\hat{y}_i|} \quad (2.30)$$

F₁ score The F₁ score is the harmonic mean between precision and recall and is the most frequently used metric when using an imbalanced dataset. It is computed as follows:

$$F_1 = \frac{1}{N} \sum_{i=1}^N 2 \cdot \frac{|y_i \cap \hat{y}_i|}{|y_i| + |\hat{y}_i|} \quad (2.31)$$

Exact Match Ratio The most strict metric is the exact match ratio (EMR) which measures the percentage of samples that have all their labels classified correctly. It is computed as given in equation 2.32 where I is the indicator function.

$$\text{EMR} = \frac{1}{N} \sum_{i=1}^N I(y_i = \hat{y}_i) \quad (2.32)$$

CHAPTER 2. RELATED WORK

Label-based metrics measure the performance of each label separately and enables to evaluate which labels perform exceptionally well or poor. This way, any known metric from binary classification can be used. The following equations specify the precision, recall and F_1 score for label k where $y_i^{(k)} = 1$ if the example contains class k and $y_i^{(k)} = 0$ otherwise. The same holds for the predictions $\hat{\mathbf{y}}$. To obtain a single score, the mean over all classes can be calculated. This corresponds to the macro average.

$$\text{Precision}^{(k)} = \frac{\sum_{i=1}^N y_i^{(k)} \hat{y}_i^{(k)}}{\sum_{i=1}^N \hat{y}_i^{(k)}} \quad (2.33)$$

$$\text{Recall}^{(k)} = \frac{\sum_{i=1}^N y_i^{(k)} \hat{y}_i^{(k)}}{\sum_{i=1}^N y_i^{(k)}} \quad (2.34)$$

$$F_1^{(k)} = \frac{2 \sum_{i=1}^N y_i^{(k)} \hat{y}_i^{(k)}}{\sum_{i=1}^N y_i^{(k)} + \sum_{i=1}^N \hat{y}_i^{(k)}} \quad (2.35)$$

2.5 Word Embeddings

Most NLP algorithms 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, i.e. it is assumed that there is no relationship between them.

2.5.1 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 2.36 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.

$$\text{TF-IDF}(w, d) = \text{TF}(w, d) \cdot \log \left(\frac{N}{\text{DF}(w)} \right) \quad (2.36)$$

2.5.2 Word2Vec

Mikolov et al. (2013) proposed Word2Vec, a technique that allows to obtain word embeddings with a neural network. The main idea is that words with similar context should be located in close spatial dimensions. There are 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 2.22a. The input of the network is a set of C one-hot encoded context words 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 learned. The idea of the skip-gram model is very similar to the CBOW, but the neural network is trained to predict the context instead of the target word (see figure 2.22b). 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 it represents frequent words better.

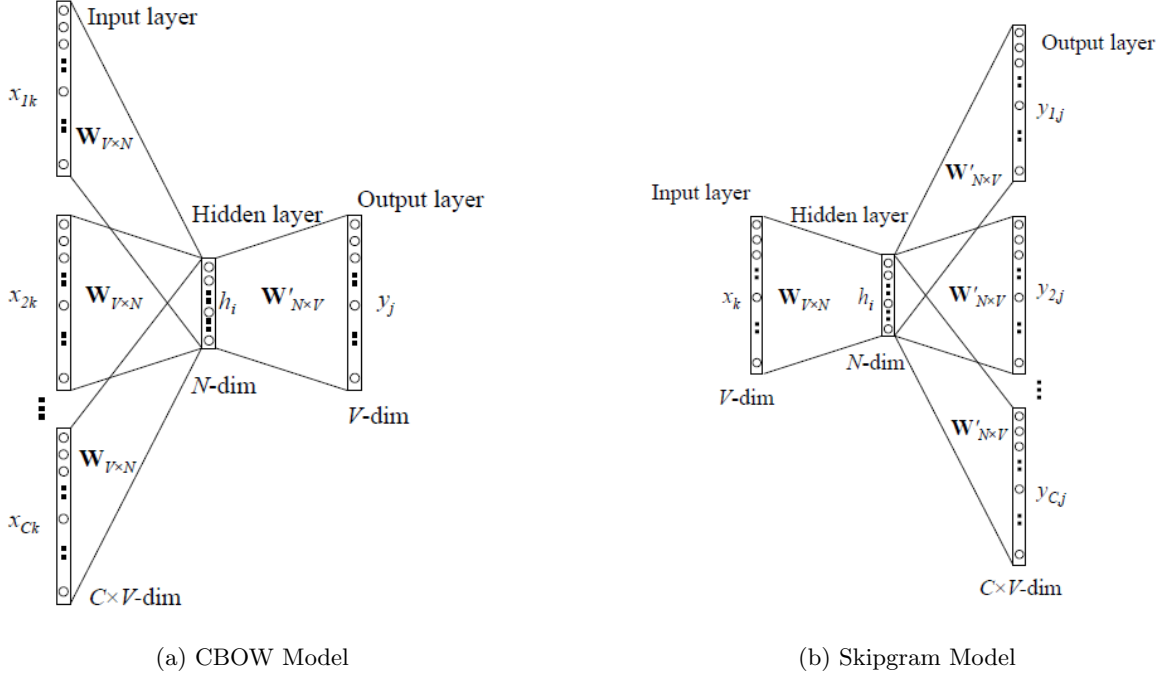


Figure 2.22: The proposed Word2Vec models, visualized by Rong (2014)

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 they could perform simple algebraic operations with these learned embeddings. For example, to find a word that is similar to *Italy* in the same sense as *France* to *Paris*, a vector \mathbf{X} can be computed as follows:

$$\mathbf{X} = \text{vector}(\text{"Paris"}) - \text{vector}(\text{"France"}) + \text{vector}(\text{"Italy"}) \quad (2.37)$$

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 cosine similarity (sim) measures the cosine of the angle θ between two vectors \mathbf{a} and \mathbf{b} and is computed as given in equation 2.38.

$$\text{sim}(\mathbf{a}, \mathbf{b}) = \cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} \quad (2.38)$$

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 solution to this question.

2.5.3 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 so that the dot product of two word vectors is equal to the logarithm of the number of times the word occurs near each other. The GloVe model minimizes the squared cost function showed in equation 2.39 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 word vectors and \mathbf{b} , $\tilde{\mathbf{b}}$ are bias terms that capture the fact that some words occur more often than others.

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

$$f(x) = \begin{cases} (x/x_{max})^\alpha & \text{if } x < x_{max} \\ 1 & \text{else} \end{cases} \quad (2.39)$$

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

2.5.4 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 word 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.

2.5.5 Flair

The approaches described above have the drawback that polysemous words are embedded into a single word vector. A polysemous word 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 to 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. The model architecture consists of an RNN with one LSTM cell. The goal of the model is to predict the next character given an input sequence of characters. Thus, 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 2.23 shows an example of the extraction of the flair embedding for the word *Washington*. Both hidden state outputs from the forward model (shown in red) and the backward model (shown in blue) are concatenated into a single word embedding.

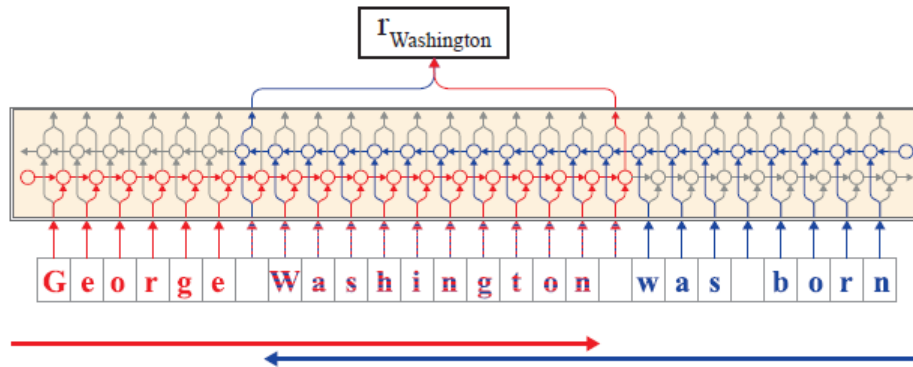


Figure 2.23: An illustration of the extraction of the flair embedding for the word *Washington* from the original flair paper by Akbik et al. (2018)

2.6 Language Models

Goldberg (2016) explains that language models describe a probability distribution over a sequence of words $\mathbf{x} = (x_1, \dots, x_T)$. They are modelled as a conditional probability of words given their previous words:

$$p(x_1, \dots, x_T) = \prod_{t=1}^T p(x_t | x_1, \dots, x_{t-1}) \quad (2.40)$$

When using neural networks, the probability distribution is often estimated in the logarithmic space. The goal of a language model is to predict the word \hat{x}_t by maximizing the following objective:

$$\arg \max_{\boldsymbol{\theta}} = \sum_{t=1}^T \log p(\hat{x}_t = x_t | x_1, \dots, x_{t-1}, \boldsymbol{\theta}) \quad (2.41)$$

where $\boldsymbol{\theta}$ represent the model parameters.

To measure the quality of a language model, a common metric is the perplexity (PPL). It measures the uncertainty of the model when predicting a word given the previous words and is computed as follows:

$$\begin{aligned} \text{PPL}(\mathbf{x}) &= \sqrt[T]{\frac{1}{p(x_1, \dots, x_T)}} \\ &= \sqrt[T]{\prod_{t=1}^T \frac{1}{p(x_t | x_1, \dots, x_{t-1})}} \end{aligned} \quad (2.42)$$

While language models are often trained to generate text conditioned on previous words, they can be extended to consider an external condition, denoted as \mathbf{c} :

$$\arg \max_{\boldsymbol{\theta}} = \sum_{t=1}^T \log p(\hat{x}_t = x_t | x_1, \dots, x_{t-1}, \mathbf{c}, \boldsymbol{\theta}) \quad (2.43)$$

Popular examples of conditional text generation are machine translation or image-to-text generation models (see section 2.7), where the external condition is the source language or the image respectively.

2.6.1 BERT Language Model

Bidirectional encoder representations from transformers (BERT) is a language model that is based on transformer networks (see section 2.7.7) and was introduced by Devlin et al. (2018). Unlike standard language models that predict the next word given the input sequence, BERT employs a masked task where 15% of the tokens are hidden. The model is then trained to predict the masked tokens. Thanks to this masking task, BERT has the unique property that it works bidirectional. To further improve the performance, they developed the next sentence prediction task. With this task, the model is trained to predict the likelihood that sentence B belongs after sentence A .

2.7 Image-to-Text Models

While multi-label image classification models identify multiple labels of an image, image-to-text models go a step further: they describe the content of an image by generating a text written in a natural language. This task is much harder since not only objects must be identified but also their relationship to one another must be expressed. Thus, it connects the field of computer vision with NLP.

2.7.1 Traditional Retrieval Approaches

Traditionally, a text is queried from a fixed-sized dataset rather than generated from a vocabulary of words. One of the earliest approaches was proposed by Farhadi et al. (2010) where they project images and sentences into a meaning space that consists of triplets. Each triplet contains an object, action and a scene. To compute the triplet of a sentence, they parse the dependencies of each sentence and extract the subject, direct object and any dependencies involving a noun and a verb. Then they use these dependencies to generate the *(object, action)* pairs. They process the head nouns of the prepositional phrases to obtain the last piece of the triplet, the scene information. To compute a triplet based on an image, they learn a multi-label Markov random field. Finally, they return all sentences that share the same triplet. Figure 2.24 depicts the proposed retrieval method.

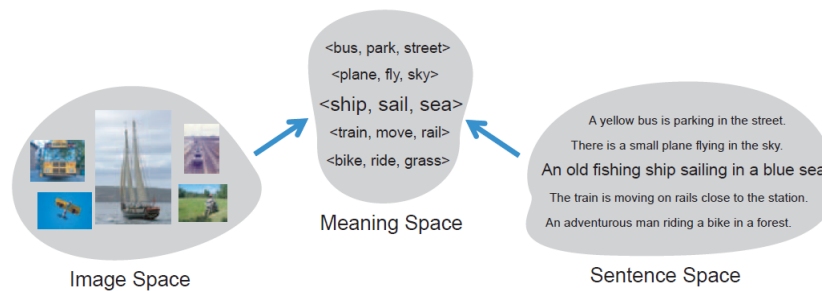


Figure 2.24: An illustration of the image-to-text retrieval method proposed by Farhadi et al. (2010)

A similar method was introduced by Kulkarni et al. (2011) where they also used template-based text generation. However, they detect multiple objects, modifiers, and their spatial relationships in order to retrieve more accurate descriptions.

These models have the disadvantage of being heavily handmade and inflexible when it involves text generation. Moreover, due to the compositional nature of language, it is unlikely that any database will contain all possible image descriptions.

2.7.2 Encoder-Decoder Models

Different from retrieval-based models, generation-based models aim to learn a language model that can generate novel descriptions with more flexible syntactical structures by sampling from a vocabulary. Vinyals et al. (2014) presented a single end-to-end model that is trained to maximize the likelihood of the description $\mathbf{y} = (y_1, \dots, y_T)$ given the input image \mathbf{I} with respect to the parameters θ :

$$p_{\theta}(y_1, \dots, y_T | \mathbf{I}) \quad (2.44)$$

Their work was inspired by neural machine translation where an input text is encoded into a fixed-sized representation by an encoder and then used by a decoder to generate the translated sequence. The difference is that they used a CNN as an encoder instead of an RNN. They treat the image-to-text task as translating an image into a text description. The encoder transforms the input image \mathbf{I} into the fixed-sized representation \mathbf{c} by feeding it into a CNN as shown in equation 2.45 where f_e denotes the encoder function.

$$\mathbf{c} = f_e(\mathbf{I}) = \text{CNN}(\mathbf{I}) \quad (2.45)$$

The decoder RNN generates the next hidden state \mathbf{h}_t by conditioning on both the context vector \mathbf{c} and the previous hidden state \mathbf{h}_{t-1} , where f_d denotes the decoder function and is usually an LSTM or GRU cell.

$$\mathbf{h}_t = f_d(\mathbf{h}_{t-1}, \mathbf{c}) = \text{RNN}(\mathbf{h}_{t-1}, \mathbf{c}) \quad (2.46)$$

Finally, the hidden state \mathbf{h}_t is fed into a linear layer and a softmax activation function to produce a probability distribution over the vocabulary words. Equation 2.47 shows how the final distribution is obtained for each step t where \mathbf{W}_v and \mathbf{b}_v are learnable parameters. Alternatively, the linear layer can be replaced by an MLP.

$$y_t \sim \mathbf{p}_t^{(\text{vocab})} = \text{softmax}(\mathbf{W}_v \mathbf{h}_t + \mathbf{b}_v) \quad (2.47)$$

Figure 2.25 visualizes the proposed architecture where the input image is encoded and then used by the decoder to compute the output text.

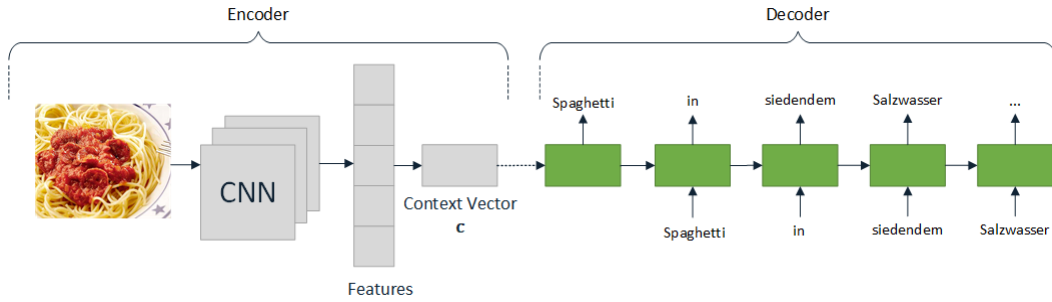


Figure 2.25: The image-to-text model proposed by Vinyals et al. (2014)

2.7.3 Encoder-Decoder Models with Attention

The previously described approach treats objects in an image the same and ignores salient objects when generating the output. Xu et al. (2015) extended the work from Vinyals et al. (2014) by not transforming the input image into a fixed-sized representation \mathbf{c} , but by using the attention mechanism to allow the decoder to attend over different parts of the input image. This should mimic the human eye focusing on different regions in an image when generating the output words.

First, the encoder CNN extracts L visual annotation vectors $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_L)$. They are then used to compute an alignment score \mathbf{e} by passing them into an alignment model. Equation 2.48 shows an alignment model that was proposed by Bahdanau et al. (2014) in the context of machine translation where \mathbf{W}_a and \mathbf{W}_b are learnable parameters.

$$e_{t,i} = \text{score}(\mathbf{h}_{t-1}, \mathbf{x}_i) = \mathbf{W}_a^\top \tanh(\mathbf{W}_b[\mathbf{x}_i; \mathbf{h}_{t-1}]) \quad (2.48)$$

These alignment scores \mathbf{e} are then transformed into attention weights α by passing them into a softmax function. Equation 2.49 states how the attention weight of the i -th image region at time step t is calculated.

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^L \exp(e_{t,k})} \quad (2.49)$$

The attention weights are then used at each time step to compute a new context vector \mathbf{c}_t as follows:

$$\mathbf{c}_t = \sum_{i=1}^L \alpha_{t,i} \cdot \mathbf{x}_i \quad (2.50)$$

This vector \mathbf{c}_t is then fed into the RNN decoder f_d along with the previous hidden state \mathbf{h}_{t-1} :

$$\mathbf{h}_t = f_d(\mathbf{h}_{t-1}, \mathbf{c}_t) \quad (2.51)$$

Figure 2.26 shows the proposed architecture that contains a special attention layer which allows the model to look at different parts of the image. This architecture has not only the advantage of improving the model's performance, but also supports interpretability. It gives a better understanding of how the model generated the output.

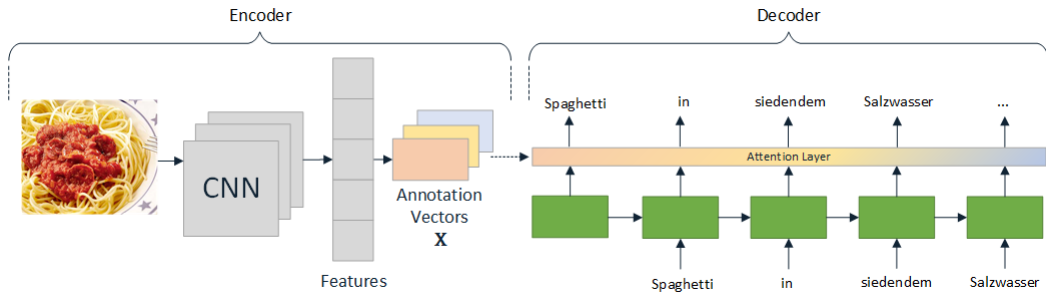


Figure 2.26: The image-to-text model with attention proposed by Xu et al. (2015)

Adaptive Attention

Lu et al. (2016) describe that classic attention-based image-to-text model force the decoder to pay attention to the input image even though in some cases it is not necessarily required. A typical example is the prediction of non-visual words such as *the* or *of*. The authors explain that gradients from non-visual words could mislead and diminish the overall effectiveness of the generation process. Therefore, they propose an adaptive encoder-decoder model that can automatically decide when to rely on visual signals and when to just rely on the language model. To do this, they extend the decoder LSTM to produce an additional *visual sentinel* vector \mathbf{s}_t . This vector is calculated as given in equation 2.52, where \mathbf{C}_t is the memory cell of the LSTM (see section 2.3.1), \mathbf{y}_t the current output, and \mathbf{W}_s a learnable parameter.

$$\mathbf{s}_t = \sigma(\mathbf{W}_s[\mathbf{y}_t; \mathbf{h}_{t-1}]) \odot \tanh(\mathbf{C}_t) \quad (2.52)$$

Subsequently, the authors calculate a new context vector $\hat{\mathbf{c}}_t$ which is computed as a linear combination between the sentinel vector \mathbf{s}_t and the visual context vector \mathbf{c}_t weighted by β_t .

$$\hat{\mathbf{c}}_t = \beta_t \mathbf{s}_t + (1 - \beta_t) \mathbf{c}_t \quad (2.53)$$

The parameter β_t is a scalar variable that indicates how much the model needs to pay attention to the image or the language model at time step t . It is the last element of the modified attention weights $\hat{\boldsymbol{\alpha}}$. They are a combination of the attention distribution over the image and the visual sentinel vector and computed as follows:

$$\begin{aligned} \hat{e}_{t,i} &= \mathbf{W}_{\hat{a}}^\top \tanh(\mathbf{W}_{\hat{b}}[\mathbf{s}_{t-1}; \mathbf{h}_{t-1}]) \\ \hat{\boldsymbol{\alpha}}_t &= \text{softmax}([\mathbf{e}_t; \hat{\mathbf{e}}_t]) \\ \beta_t &= \hat{\alpha}_{t,L+1} \end{aligned} \quad (2.54)$$

where $\mathbf{W}_{\hat{a}}$ and $\mathbf{W}_{\hat{b}}$ are again learnable parameters. Finally, to compute the vocabulary distribution, they extend equation 2.47 by adding the combined context vector $\hat{\mathbf{c}}_t$ to the hidden state \mathbf{h}_t as follows:

$$y_t \sim \mathbf{p}_t^{(\text{vocab})} = \text{softmax}(\mathbf{W}_v(\mathbf{h}_t + \hat{\mathbf{c}}_t) + \mathbf{b}_v) \quad (2.55)$$

A related approach has been proposed by Wei et al. (2020). The authors explain that classic attention models have the drawback that they work either hard, i.e. the generated attention distribution will be relatively concentrated, or soft. When generating nouns, the model may require hard attention, focusing on some specific objects of the image. On the other hand, the model may require soft attention when generating conjunctions and prepositions, drawing attention to all regions of the image. To mitigate these issues, the authors proposed an adaptive attention where the model can control the focus intensity of visual attention and integrate visual information better for different generated words. They compute a coefficient η_t that controls the focus intensity:

$$\begin{aligned} \beta_t &= \tanh(\mathbf{W}_\beta[\bar{\mathbf{X}}; \mathbf{h}_{t-1}]) \\ \eta_t &= \lambda^{\beta_t} \end{aligned} \quad (2.56)$$

where $\bar{\mathbf{X}}$ are the average image features, \mathbf{W}_β a learnable parameter and λ a hyperparameter. Finally, they use η_t to weight the alignment scores \mathbf{e} while generating the attention weights $\boldsymbol{\alpha}$:

$$\alpha_{t,i} = \frac{\exp(\eta_t e_{t,i})}{\sum_{k=1}^L \exp(\eta_t e_{t,k})} \quad (2.57)$$

Semantic Attention

You et al. (2016) describe that image-to-text approaches work either top-down by predicting multiple attributes of an image and then converting them to words (see section 2.7.1), or bottom-up by generating words that describe various aspects of an image and then combining them. The authors introduced a method that makes use of both approaches. First, they use an encoder CNN to extract a context vector \mathbf{c} of a given image. They describe that this vector is only used to give the decoder a quick overview of the image content. Thus, they use it to initialize the first hidden state \mathbf{h}_0 of the decoder RNN. Additionally, they train a multi-label classification model to predict a set of image attributes $\mathbf{A} = \{\mathbf{a}_1, \dots, \mathbf{a}_M\}$. They determine the different classes by selecting the most common words from the training data. Finally, the decoder uses these attributes with the attention mechanism to generate the output text. Since this architecture exploits not only an overview understanding of the image but also abundant fine-grain visual semantic aspects, the authors call it semantic attention.

He and Hu (2019) extended this method by allowing the decoder model to also pay attention to the image annotation features \mathbf{X} and thus calling their architecture *semantic double attention*.

A similar approach was proposed by Jing et al. (2018) where they first generate a context vector $\mathbf{c}_t^{(X)}$ based on the visual features \mathbf{X} and then a context vector $\mathbf{c}_t^{(A)}$ based on the image attributes \mathbf{A} . They then concatenate these two vectors and feed them into a fully connected layer \mathbf{W}_{fc} to obtain the final context vector \mathbf{c}_t :

$$\mathbf{c}_t = \mathbf{W}_{fc}[\mathbf{c}_t^{(X)}; \mathbf{c}_t^{(A)}] \quad (2.58)$$

Figure 2.27 visualizes the semantic attention model where the encoder generates the annotation vectors \mathbf{X} and image attributes \mathbf{A} , allowing the decoder to pay attention to both vectors to compute the output.

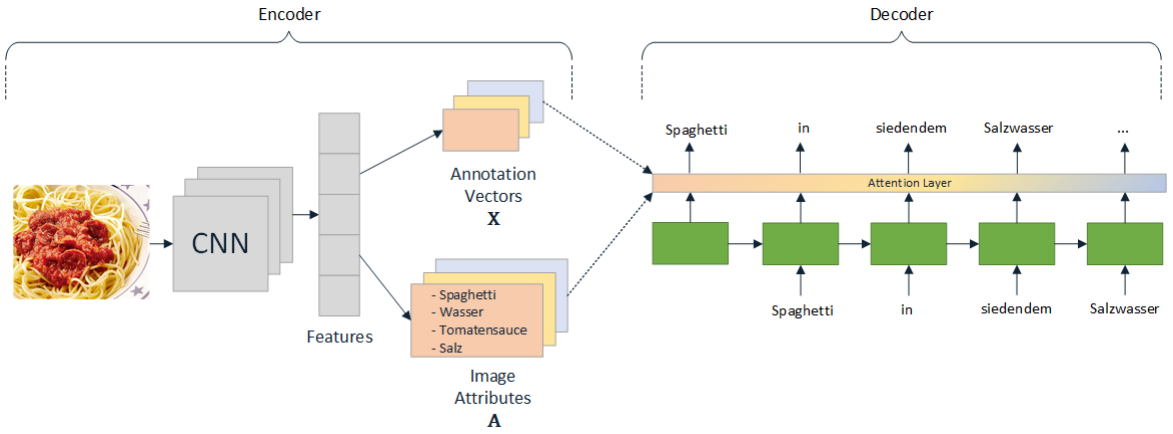


Figure 2.27: The image-to-text model with semantic attention

2.7.4 Dense Captioning

The previously described approaches share the limitation that the generated output lacks in detail. They are only capable of describing images with a single high-level sentence. Johnson et al. (2016) introduced the dense captioning task which requires a model to predict a set of descriptions across regions of an image. To achieve this task, they proposed a model that combines object detection with natural language description. It overcomes the limitation of generating single high-level sentences by detecting multiple regions of interest in an image and describing each with a short phrase. An overview of the proposed model is depicted in figure 2.28. The authors first extract features of the input image by passing it to a VGG-16 network. These features are then fed into their proposed fully convolutional localization layer which was inspired by the Faster R-CNN object detection model (Ren et al., 2015). Each detected region feature is passed into the recognition network that compactly encodes its visual appearances. Finally, the LSTM is conditioned on these region codes to produce the output text for each of the detected objects.

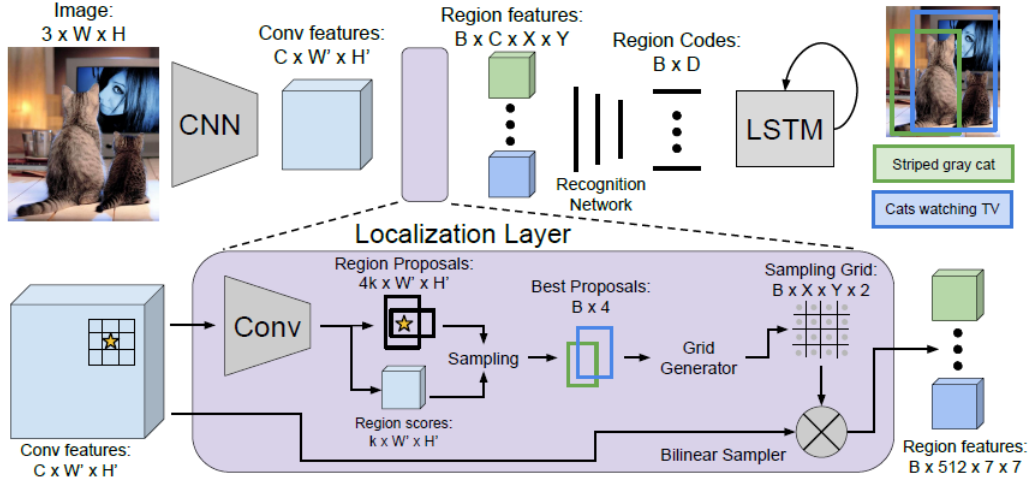


Figure 2.28: The architecture of the proposed dense captioning model by Johnson et al. (2016)

2.7.5 Hierarchical Approach

Although dense captioning allows to describe an image more in detail, it has the main drawback that the descriptions are often not coherent, i.e. they do not form a cohesive whole that describes the entire image since all descriptions are independent. Krause et al. (2017) addressed this shortcoming by introducing the task of generating paragraphs that richly describe images. Figure 2.29 illustrates their proposed model. The input image is passed to a region detector to identify objects and other regions of interest. The region features are aggregated to produce a pooled representation richly expressing the image semantics. This representation is then taken as input to a hierarchical RNN that contains two levels: a sentence RNN and a word RNN. The sentence RNN predicts how many sentences to generate and produces an input topic vector for each sentence. Finally, the word RNN is initialized with these topic vectors to generate the output text.

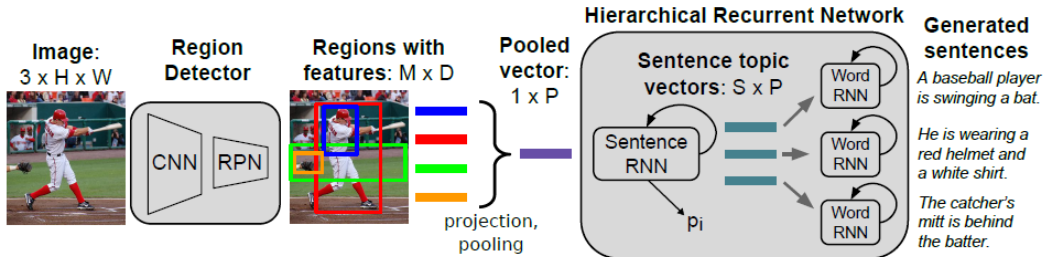


Figure 2.29: An overview of the proposed model by Krause et al. (2017)

2.7.6 CNN+CNN Models

The aforementioned methods are all under the framework of CNN+RNN where a CNN is used to extract the image features and a RNN to generate the output text. Although they provide satisfying results, they have the disadvantage that they cannot be parallelized, since RNNs must be calculated step-by-step. To generate the current hidden state \mathbf{h}_t the previous hidden state \mathbf{h}_{t-1} is required as input.

Wang and Chan (2018) replaced the RNN decoder with a CNN. Their architecture is composed of four modules: a vision module to extract features from an image, a language module to model sentences, an attention module to connect the vision and language module, and lastly a prediction module that uses the features to predict the next word. The vision module consists of a VGG-16 network without any fully-connected layers to extract L visual features $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_L)$. The language model is based on a CNN without the use of pooling layers. They project a sentence with N words into an embedding space $\mathbf{E} = (\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_N)$ and feed them into a stack of convolutional layers with gated linear units (GLUs). Algorithm 9 shows how the output \mathbf{h} at layer l is computed, where $\mathbf{W}_a^{[l]}$ and $\mathbf{W}_b^{[l]}$ denote the kernel of the l -th layer, while $\mathbf{b}_a^{[l]}$ and $\mathbf{b}_b^{[l]}$ are bias vectors. Moreover, the operation \star is the convolutional operator and σ the sigmoid activation function. They initialize $\mathbf{h}^{[0]}$ with \mathbf{E} .

Algorithm 9 Compute the output of the CNN decoder using GLU activations

$$\mathbf{h}_a^{[l]} \leftarrow \mathbf{W}_a^{[l]} \star \mathbf{h}^{[l-1]} + \mathbf{b}_a^{[l]}$$

$$\mathbf{h}_b^{[l]} \leftarrow \mathbf{W}_b^{[l]} \star \mathbf{h}^{[l-1]} + \mathbf{b}_b^{[l]}$$

$$\mathbf{h}^{[l]} \leftarrow \mathbf{h}_a^{[l]} \odot \sigma(\mathbf{h}_b^{[l]})$$

The output of the CNN are concept vectors $\mathbf{A} = (\mathbf{a}_1, \dots, \mathbf{a}_N)$. The attention module takes the visual features \mathbf{X} and concept vectors \mathbf{A} . For each concept \mathbf{a}_t and visual feature \mathbf{v}_i an attention score $e_{t,i}$ is computed as follows:

$$e_{t,i} = \mathbf{a}_t^\top \mathbf{W} \mathbf{x}_i \quad (2.59)$$

where \mathbf{W} is a learnable parameter matrix. Then, the scores \mathbf{e} are fed into a softmax function to obtain attention weights α . Using these weights, similar to the RNN-based image-to-text models, a context vector \mathbf{c}_t is obtained by computing the weighted average between the image features \mathbf{v}_i and the attention weights $\alpha_{t,i}$ as shown in equation 2.50. The prediction module is a one-hidden layer neural network that computes the output probabilities using the context vector \mathbf{c}_t and the concept \mathbf{a}_t .

Aneja et al. (2018) proposed a similar CNN+CNN image-to-text model, however, they were using one fully-connected layer of the VGG-16 encoder network.

2.7.7 Approaches using Transformer Networks

Replacing RNNs with CNNs allows parallelization of sequence generation tasks. However, such models require a huge number of operations to relate signals from two arbitrary input or output positions. Vaswani et al. (2017) introduced the transformer model that relies entirely on the attention mechanism instead of using RNNs or CNNs. It was first proposed for machine translation and has been successfully applied to many NLP tasks. Zhu et al. (2018) adapted the transformer network by replacing the encoder transformer with a CNN to use it for image-to-text tasks. Figure 2.30 provides an overview of the transformer-based image-to-text architecture that contains a stack of N_{layers} decoders.

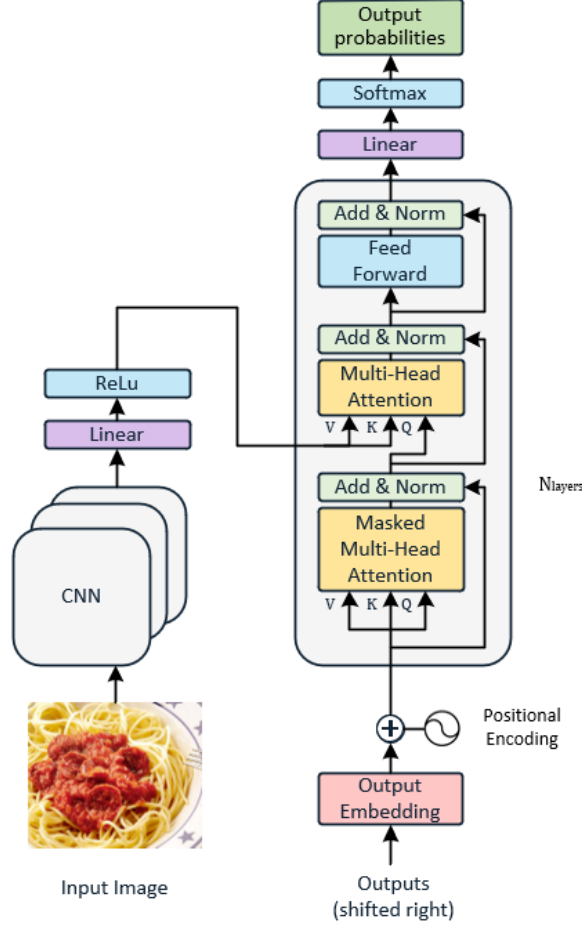


Figure 2.30: The architecture of the transformer based image-to-text model proposed by Zhu et al. (2018)

Encoder

For the encoder CNN, they use a pre-trained ResNeXt model followed by a linear layer with the ReLU activation function to extract L image regions $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_L)$, $\mathbf{x}_i \in \mathbb{R}^{d_{\text{model}}}$.

Decoder

The transformer decoder uses the multiplicative attention from Luong et al. (2015), however, they divide the output by a scaling factor. Thus, the attention method is called *scaled dot-product attention*. The input of this attention function consists of three different matrices: a query \mathbf{Q} and key \mathbf{K} matrix of dimension $d_k \times L$ as well as a value matrix \mathbf{V} of dimension $d_v \times L$. The output is computed as follows:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{d_k}} \right) \mathbf{V} \quad (2.60)$$

The queries, keys and value matrices are computed by multiplying the input with a corresponding weight matrix $\mathbf{W}^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $\mathbf{W}^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ and $\mathbf{W}^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$, which are parameters trained jointly with the model. The dot product between the key and query matrices could grow large in magnitude and thus leading to small gradients. To stabilize the gradients, the result is divided by the square root of d_k before passing it to the softmax function. This 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 2.61 and visualized in figure 2.31.

$$\begin{aligned} \text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) &= \text{Concat}(\text{head}_1, \dots, \text{head}_h) \cdot \mathbf{W}^O \\ \text{where head}_i &= \text{Attention}(\mathbf{Q} \cdot \mathbf{W}_i^Q, \mathbf{K} \cdot \mathbf{W}_i^K, \mathbf{V} \cdot \mathbf{W}_i^V) \end{aligned} \quad (2.61)$$

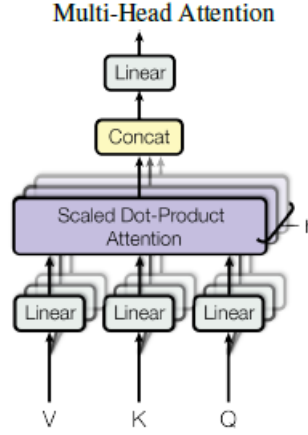


Figure 2.31: A visualization of the computation graph of the multi-head attention in Vaswani et al. (2017)

Each of the N_{layers} decoder layers applies the multi-head attention function two times: using the currently generated output \mathbf{y} (masked self-attention) and the encoded image features \mathbf{V} (encoder-decoder attention).

Masked Self-Attention Each word of the output $\mathbf{y} = (y_1, \dots, y_N)$ is projected into a d_{model} -dimensional vector by using an embedding layer. All of these embedding vectors in one sentence are combined into a matrix of shape $L \times d_{\text{model}}$ and are fed into the decoder. To account for the lack of recurrence in the transformer network, the input embeddings are positionally encoded. This is done by injecting information about the relative position of the words in the sequence. The encodings are sine and cosine functions of different frequencies and are added to the input embeddings. It has the effect that locations have similar position-encoding vectors. Equation 2.62 shows how these encoding are computed, where pos is the position and i the dimension.

$$\begin{aligned} \text{PE}_{(\text{pos}, 2i)} &= \sin(\text{pos}/10000^{2i/d_{\text{model}}}) \\ \text{PE}_{(\text{pos}, 2i+1)} &= \cos(\text{pos}/10000^{2i/d_{\text{model}}}) \end{aligned} \quad (2.62)$$

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. Moreover, this attention layer uses masks to prevent the model from seeing future information. This is done by masking subsequent positions, i.e. setting them to $-\infty$, before applying the softmax function in the attention calculation. It ensures that the model generates the current word by only using the previously seen words.

Encoder-Decoder Attention The latter use-case of attention is called encoder-decoder attention where both the value and key matrix are the image features \mathbf{X} generated by the encoder. The query matrix is the previously generated output after applying masked self-attention. The usage of this kind of attention is similar to the one applied in traditional RNN-based image-to-text models: to make the relation between the spatial information of the image and the output sentence.

Between each module of the decoder, there are residual connections which are used to retain the positional encodings. In order to reduce the training time and to stabilize the hidden states, layer normalization is performed after each layer. 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 $\mathbf{p}_t^{(\text{vocab})}$.

Multi-Level Supervision

Additionally, the authors propose a new multi-level supervision mechanism to leverage the multi-layer outputs of the Transformer decoder layers. At every layer, they compute the output sequence by projecting the decoder output into a vocabulary distribution with a linear layer. They then compute and minimize the loss of each of these intermediate outputs as depicted in figure 2.32. At inference time, they use an average pooling layer to combine all outputs.

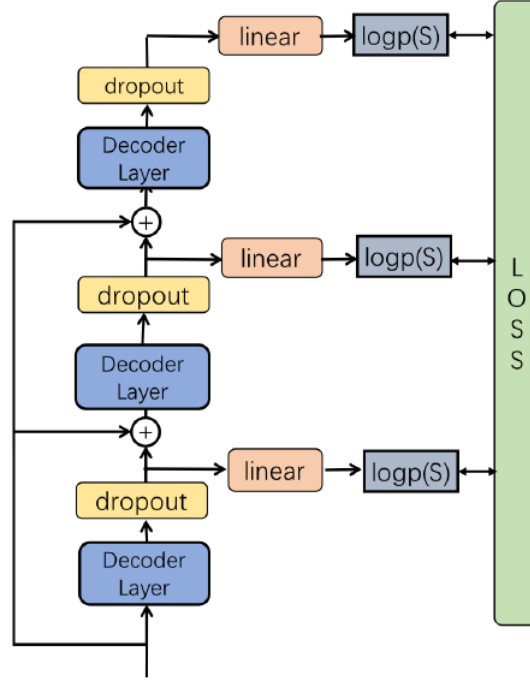


Figure 2.32: Multi-level supervision for three transformer decoder layers proposed by Zhu et al. (2018)

Springenberg et al. (2018) used a similar architecture, however, they extended the image-encoder CNN with self-attention which allows for better feature representations. Yu et al. (2019) replaced the ResNeXt image-encoder with a Faster R-CNN object detection model to extract image regions.

2.7.8 Decoding Strategies

Regardless of using RNNs, CNNs or transformer models, the aforementioned architectures have something in common: they all use an autoregressive decoder, i.e. generating a word is conditioned on the previously generated words. For each word of the output, the decoder predicts a multinomial probability distribution over the vocabulary. The goal is to maximize the joint probability of the target sequence \mathbf{y} , given an image \mathbf{I} :

$$p(\mathbf{y}|\mathbf{I}) = \prod_{t=1}^T p(y_t|y_1, \dots, y_{t-1}, \mathbf{I}) \quad (2.63)$$

To produce an output text, 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. The disadvantage of this strategy is that it only maximizes the probability for each next word in isolation. Thus it can lead to a bad result as soon as the model makes a mistake in a single step.

The second approach is called beam search. Sutskever et al. (2014) use a beam search decoder that does not only select the word with the highest probability but also keeps track of B partial hypotheses. At each time step, every partial hypothesis in the beam is extended with all words in the vocabulary. Subsequently, 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 only the hypothesis with the highest score is kept. 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 greedy search. The parameter beam size B is a hyperparameter that has to 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 2.33 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 $\langle \text{START} \rangle$ *Spaghetti in Salzwasser kochen* . $\langle \text{END} \rangle$ achieves the highest sum of log-probabilities and is therefore selected.

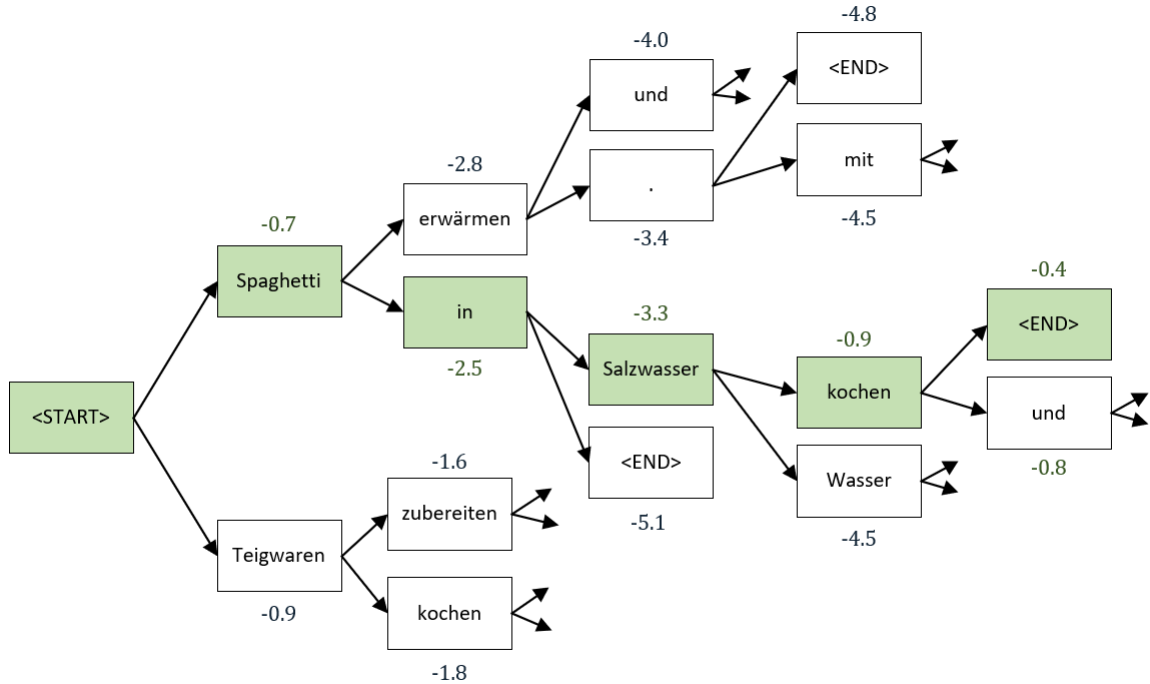


Figure 2.33: An example of a beam search decoder with beam size 2

Wu et al. (2016) refined the pure maximum probability beam search by introducing a length normalisation. They show that beam search favours shorter outputs over longer ones since adding a negative log-probability decreases the score of the hypothesis. Equation 2.64 states how the score of each hypothesis is normalised by dividing it by a length penalty $\text{lp}(\mathbf{y})$. The authors suggest to use an α value between 0.6 and 0.7 for the length penalty.

$$\begin{aligned}\text{score}(\mathbf{y}, \mathbf{I}) &= \frac{\log(p(\mathbf{y}|\mathbf{I}))}{\text{lp}(\mathbf{y})} \\ \text{lp}(\mathbf{y}) &= \frac{(5 + |\mathbf{y}|)^\alpha}{(5 + 1)^\alpha}\end{aligned}\tag{2.64}$$

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 $n = 2$.

2.7.9 Metrics

Evaluation is crucial for measuring progress and encouraging state-of-the-art improvements. Classification tasks where the output is a label are straightforward to evaluate. By means of simple label matching to compute the confusion matrix, metrics like accuracy, precision, or recall can be calculated. However, evaluating image-to-text tasks is much more complex. Authors of such models often disregard the image and pose the evaluation as a purely linguistic task similar to machine translation or text summarization evaluation. Various metrics for evaluating image-to-text tasks are presented below where \hat{y} denotes the prediction and y a set of reference descriptions.

BLEU Score

The bilingual evaluation understudy (BLEU) score was introduced by Papineni et al. (2002) as a machine translation evaluation metric that is quick and language-independent. They measure the closeness to one or more reference texts according to a numerical metric. The main challenge is that there are many "perfect" output sentences 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 -grams which occur in any reference corpus and then divide it by the total number of words in the translation. Table 2.1 shows an example where the unigram precision is $6/6 = 1$, because each of the 6 words in the prediction appears in the reference text. This is clearly not a good evaluation metric. Therefore, 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. Next, the total count of each predicted word is clipped by its maximum reference count. These clipped counts are added up and then divided by the total number of words. The modified unigram precision for the example in table 2.1 is now $1/6$ instead of $6/6$.

References y	Prediction \hat{y}
Spaghetti in siedendem Salzwasser <u>kochen</u> Wasser <u>kochen</u> und Spaghetti dazugeben	<u>kochen</u> <u>kochen</u> <u>kochen</u> <u>kochen</u> <u>kochen</u> <u>kochen</u>
Unigram precision = $6/6$	Modified unigram precision $p_n = 1/6$

Table 2.1: BLEU score example for unigram precision

The authors have shown that p_n decays exponentially with n . Yet in all cases, they were able to distinguish between a good and a bad prediction. Therefore, they combine multiple p_n to a single number. Equation 2.65 shows how the modified n -gram precision is calculated for an arbitrary size of n where \hat{y} is the output of the translation 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.

$$\begin{aligned} \text{Count}_{\text{clip}} &= \min(\text{Count}, \text{MaxRefCount}) \\ p_n &= \frac{\sum_{\text{gram}_n \in \hat{y}} \text{Count}_{\text{clip}}(\text{gram}_n)}{\sum_{\text{gram}'_n \in \hat{y}} \text{Count}(\text{gram}'_n)} \end{aligned} \quad (2.65)$$

Predictions 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 2.66, the brevity penalty factor (BP) is 1 if the length of the prediction $l_{\hat{y}}$ is larger than the length of the reference l_y and exponentially decaying in $l_y/l_{\hat{y}}$ otherwise.

$$\text{BP} = \begin{cases} 1 & \text{if } l_{\hat{y}} > l_y \\ e^{1-l_y/l_{\hat{y}}} & \text{if } l_{\hat{y}} \leq l_y \end{cases} \quad (2.66)$$

The authors used multiple n -grams and averaged them with a geometric mean. Equation 2.67 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 $w_n = 1/N$.

$$\text{BLEU}_N = \text{BP} \cdot \exp \left(\sum_{n=1}^N w_n \log(p_n) \right) \quad (2.67)$$

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. There are four different types of metrics: N, L, W and S. In the following section the ROUGE-N score is described and therefore referred as 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 of the output appear in the references, the ROUGE score counts how many n -grams of the reference appear in the output.

$$\text{ROUGE}_n \text{ Recall} = \frac{\sum_{y_i \in \{\mathbf{y}\}} \sum_{\text{gram}_n \in y_i} \text{Count}_{\text{match}}(\text{gram}_n)}{\sum_{y_i \in \{\mathbf{y}\}} \sum_{\text{gram}_n \in y_i} \text{Count}(\text{gram}_n)} \quad (2.68)$$

Equation 2.68 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 model is really long, the ROUGE score would be high, 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 2.69.

$$\text{ROUGE}_n \text{ Precision} = \frac{\sum_{y_i \in \{\mathbf{y}\}} \sum_{\text{gram}_n \in y_i} \text{Count}_{\text{match}}(\text{gram}_n)}{\sum_{\text{gram}_n \in \hat{\mathbf{y}}} \text{Count}(\text{gram}_n)} \quad (2.69)$$

With the precision and recall at hand, the ROUGE_n F_1 score is calculated as the harmonic mean between the recall and precision:

$$\text{ROUGE}_n F_1 = 2 \cdot \frac{\text{ROUGE}_n \text{ Precision} \cdot \text{ROUGE}_n \text{ Recall}}{\text{ROUGE}_n \text{ Precision} + \text{ROUGE}_n \text{ Recall}} \quad (2.70)$$

METEOR Score

Both metrics BLEU and ROUGE do not account for the word order. 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.

CIDEr Score

The aforementioned metrics were all designed for other tasks such as machine translation or text summarization. Vedantam et al. (2015) proposed consensus-based image description evaluation (CIDEr), an evaluation metric specifically designed for image-to-text tasks that can be used to measure how well a prediction sentence \hat{y} matches the consensus of a set of image descriptions \mathbf{y} . To calculate the score, all words (both predictions and references) are transformed into their root form by applying stemming. For each of the stemmed sentences, the n -gram representations are obtained. The authors explain that n -grams which commonly occur across all images in the dataset should be given lower weight, since they are less informative in describing the content of an image. To do this, they perform a TF-IDF weighting for each n -gram in the whole corpus.

The CIDEr_n score for n -grams of length n is computed using the average cosine similarity between the predicted sentence and the reference sentences as given in equation 2.71. Symbol $\mathbf{g}^{[n]}$ denotes the TF-IDF vectors corresponding to all n -grams of length n , and $\|\mathbf{g}^{[n]}\|$ is its magnitude.

$$\text{CIDEr}_n = \frac{\mathbf{g}^{[n]}(\hat{y}) \cdot \mathbf{g}^{[n]}(\mathbf{y})}{\|\mathbf{g}^{[n]}(\hat{y})\| \cdot \|\mathbf{g}^{[n]}(\mathbf{y})\|} \quad (2.71)$$

Similarly to the BLEU score, the final CIDEr score is calculated by a weighted average of multiple n -gram scores as shown in equation 2.72. The authors suggest uniform weights $w_n = 1/N$ and $N = 4$ as default parameters.

$$\text{CIDEr} = \sum_{n=1}^N w_n \text{CIDEr}_n \quad (2.72)$$

Vedantam et al. (2015) additionally presented an adaption of the CIDEr metric which they refer as CIDEr-D. In contrast to the standard formulation, they do not stem the words to compute the score. They introduced an additional penalty based on the difference between the length of the prediction description and the reference lengths. Finally, they clip the number of n -gram occurrences of the predictions to the number of reference occurrences as in the BLEU score. This score is often used for image captioning challenges such as the annual COCO challenge (Chen et al., 2015). The authors explain that this adapted score is more suited for online challenges since it is better protected against gameability. The "gaming" of a metric refers to implementing algorithms that directly optimize such metrics in order to win prize money from an online challenge. Even though an algorithm would achieve a high score, it would produce poor results when judged by humans.

SPICE Score

All aforementioned metrics have something in common: they measure n -gram overlaps. Anderson et al. (2016) explain that a n -gram overlap is neither necessary nor sufficient for two sentences to convey the same meaning. Therefore, they proposed semantic propositional image captioning evaluation (SPICE) as an alternative evaluation metric for image captioning problems. The main idea is that both candidate and reference sentences are transformed in a graph-based semantic representation, which they refer as *scene graph*. It explicitly encodes the objects, attributes and relationships found in the sentences, abstracting away most of the lexical and syntactic properties of natural language. Given this graph, bigrams tuples are extracted, containing one, two or three elements, representing objects, attributes and relations respectively. The scene graph depicted in 2.34 is converted into the following tuples:

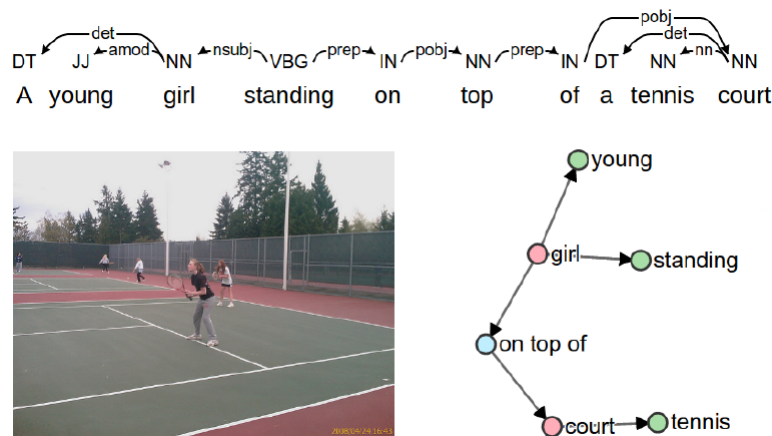
$$\{(girl), (court), (girl, young), (girl, standing), (court, tennis), (girl, on-top-of, court)\}$$


Figure 2.34: An example where a sentence is transformed into a semantic scene graph, illustrated in Anderson et al. (2016)

Similar to the ROUGE score, the precision and recall scores are calculated by matches between the candidate and reference tuples. To generate matches, synonyms are considered using the WordNet lexical database, following a similar procedure as in the METEOR score. Finally, an F_1 score is calculated using the precision and recall scores.

Word Movers Distance

Kilickaya et al. (2017) explain that two sentences, even though they do not have the same words or synonyms, can be semantically similar. On the other hand, two sentences with similar objects, attributes or relationships might not be semantically similar. To address this, they propose to use the word movers distance (WMD) for evaluating image captioning systems. The WMD was proposed by Kusner et al. (2015) to measure the distance of two documents. They compute a word travel cost $c(i, j)$ between two words i and j which corresponds to the Euclidean distance between their embedding representations \mathbf{x}_i and \mathbf{x}_j :

$$c(i, j) = \|\mathbf{x}_i - \mathbf{x}_j\|_2 \quad (2.73)$$

Then, the WMD is defined as the minimum cumulative cost required to move all word embeddings of one document to aligned word embeddings of the other document. In comparison to other metrics, this formulation completely ignores word order and readability but considers semantically similar words. Despite its usefulness, it has the drawback of being relatively expensive to calculate. Its time complexity is $\mathcal{O}(n^3 \log n)$, where n is the number of unique words. The relaxed WMD was proposed by Kusner et al. (2015) as an alternative to the WMD which has a time complexity of $\mathcal{O}(n^2)$. Figure 2.35 illustrates the WMD for two documents. All non-stop words of both documents are encoded into an embedding space. The distance between the two documents is the minimum cumulative distance that all words in document 1 need to travel to exactly match the document 2.

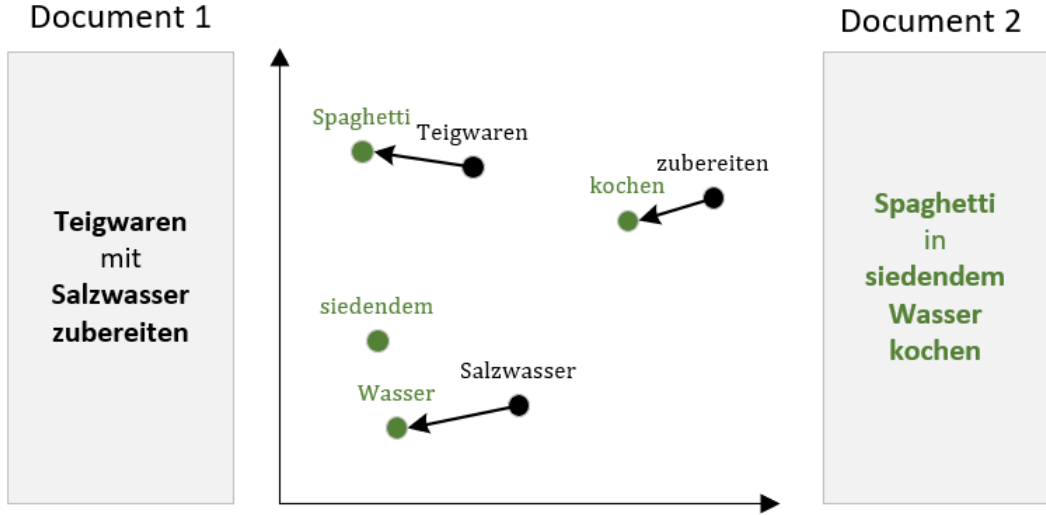


Figure 2.35: An illustrations of the WMD metric

Kilickaya et al. (2017) uses this distance metric to evaluate their image-to-text model by converting the WMD into a word movers score (WMS) by calculating the negative exponential of the WMD as follows:

$$\text{WMS} = -\exp(\text{WMD}(y, \hat{y})) \quad (2.74)$$

2.7.10 Training

Image-to-text models are traditionally trained using the cross entropy loss and evaluated using discrete non-differentiable NLP metrics such as BLEU or CIDEr. More specifically, they are trained to maximize the likelihood of the next word given the previous ground truth words. The approach of passing the ground truth at each time step is known as teacher-forcing (Williams and Zipser, 1989) and increases the effectiveness of the training. However, this creates a mismatch between training and testing, since at testing-time the model uses the previously generated words to predict the next word. This mismatch is called exposure bias. Schmidt (2019) describes that it can result in error accumulation during inference time since the model has never been exposed to its own predictions. Figure 2.36 compares the free running with the teacher forcing approach.

There are several ways of mitigating the exposure bias and two of the most commonly used approaches are explained in the following.

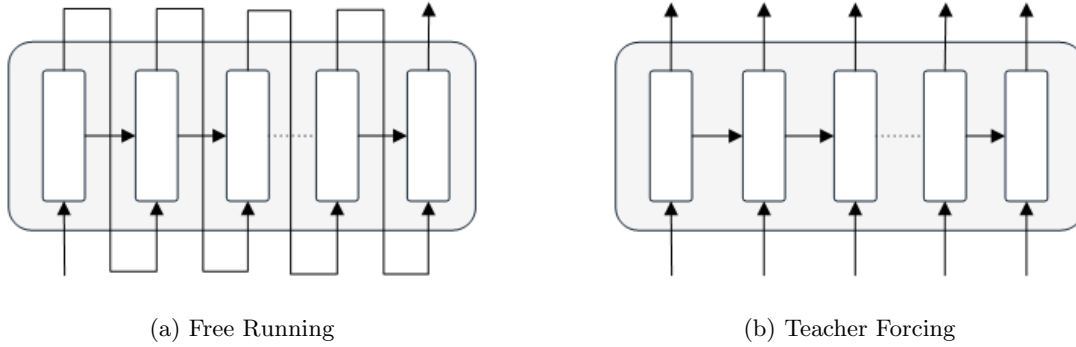


Figure 2.36: Comparison between teacher forcing during training and free running during inference time

Scheduled Sampling An approach of mitigating the exposure bias was described by Bengio et al. (2015). The authors proposed a sampling technique to randomly decide during training whether the ground truth token y_{t-1} or the prediction \hat{y}_{t-1} should be used to compute the next hidden state. The hyperparameter ϵ denotes the probability of taking the ground truth value, i.e. $\epsilon = 1$ is equal to teacher forcing. However, this does not work for transformer networks since to generate the next word, the model is conditioned on all previous words in the sequence and not just the last word. Mihaylova and Martins (2019) adapted the transformer architecture by introducing a two-pass decoding strategy that allows to use the scheduled sampling method for transformers. Firstly, the decoder computes the outputs using standard teacher forcing. Secondly, the outputs are mixed with the ground truth sequence to obtain a new reference sequence, imitating the scheduled sampling method. Lastly, the decoder is called again with teacher forcing, however, using the new reference sequence.

Professor Forcing Goyal et al. (2016) proposed professor forcing, a technique that uses the framework from generative adversarial networks (GANs) (Goodfellow et al., 2014) to encourage the dynamics of the model to be the same when training conditioned on ground truth words and when predicting freely during inference time. They train a discriminator that is tasked to distinguish between outputs generated using teacher forcing or in the free running mode. This way the distributions of hidden states are encouraged to be close to each other. The approach is illustrated in figure 2.37.

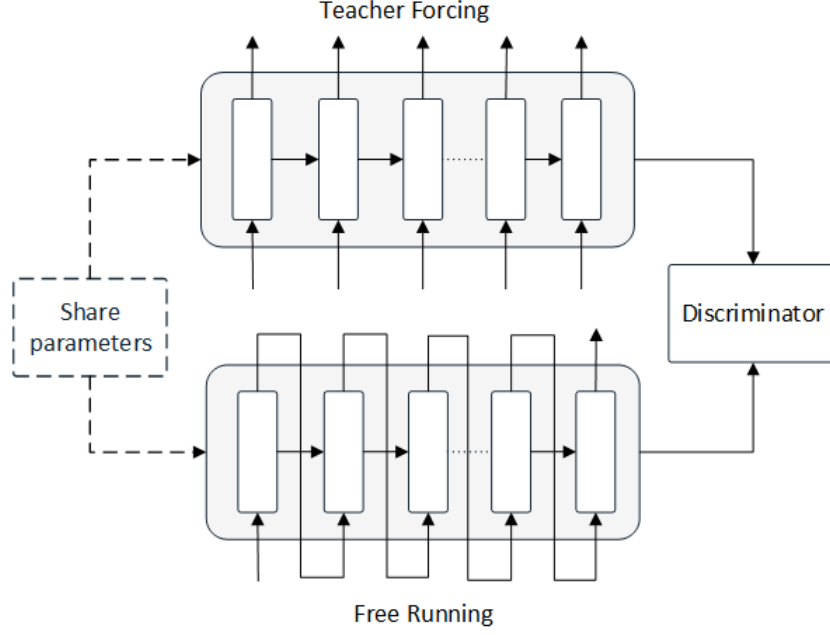


Figure 2.37: An illustration of the professor forcing technique proposed by Goyal et al. (2016)

Self-Critical Sequence Training Inspired by the recent success of reinforcement learning, Rennie et al. (2016) proposed a new approach to improve the performance of sequence-generation models which they call self-critical sequence training (SCST). They first train a sequence-generation model using the standard cross entropy loss and teacher forcing. After training, they apply the SCST training procedure to directly optimize NLP metrics and thus addressing the exposure bias issue. They view the decoder as an *agent* that interacts with an external *environment* which are words and image features. The parameters of the model θ define a policy p_{θ} that is used to select an optimal *action* which is the prediction of the next word. Upon generating the end-of-sequence token, the agent receives a *reward* r which is a NLP metric such as CIDEr.

2.8 Inverse Cooking

In the kitchen, we rely on recipes from cooking books or websites. They provide us with instructions on how to cook a given dish, which we usually find by its name. However, if we eat in a restaurant, there is no way to access detailed information about how the dish was prepared. Moreover, food culture is spreading more than ever in the current digital age as many people share food pictures in social media. This huge amount of data inspired researchers to gather food images and making datasets publicly available. Bossard et al. (2014) presented the Food-101 dataset consisting of 101'000 images of dishes from a total of 101 categories. Figure 2.38 shows examples from their introduced dataset. Such datasets have enabled significant advances in food recognition by providing reference benchmarks to experiment with new image classification models. As a result, there is now extensive research in building inverse cooking systems capable of identifying food categories and ingredients or even generating whole recipes. This section reviews the possibilities of using machine learning in the domain of cooking.

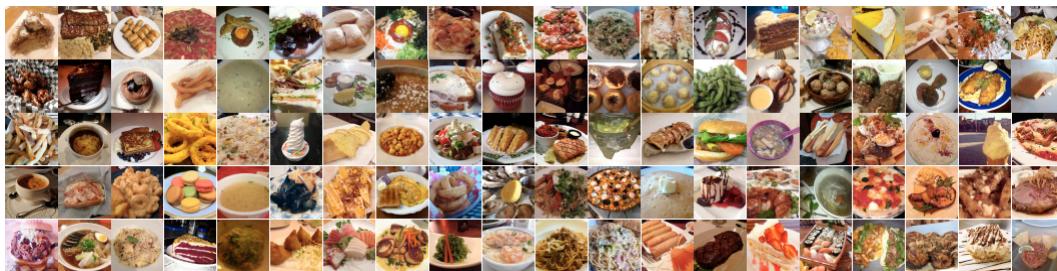


Figure 2.38: Examples from the Food-101 dataset by Bossard et al. (2014)

2.8.1 Recipe Retrieval

Salvador et al. (2017) introduced Recipe1M, a structured corpus over 1 million recipes written in English, paired with 800'0000 of food images. The authors prepared this dataset by crawling several cooking websites. As a result, they obtained a list of ingredients, a list of recipe instructions and optionally an image for each recipe. To separate the names from the ingredients, they trained a bidirectional LSTM network that performs logistic regression on each word with manually labelled data. Based on this dataset, the authors trained a retrieval model that allows a user to query a recipe given an input image. The model provides the recipe with the highest similarity to the input image. In order to calculate similarities, both the recipes and the images need to be projected into an embedding space.

Representation of Recipes To represent the list of ingredients in the embedding space, the authors trained a Word2Vec model to obtain word vectors for each ingredient. They then feed them into a bidirectional LSTM network to generate a fixed-sized vector. Since the recipe descriptions are quite long, using a single RNN could lead to vanishing gradients. Therefore, they divide each description into multiple instructions and then feed them into an LSTM network to obtain a fixed-sized representation. These encoded instructions are then fed into a separate LSTM network and its result is concatenated with the encoded ingredients. Finally, those embeddings are fed into a fully connected layer to obtain the recipe representation.

Representation of Images The authors use a pre-trained ResNet-50 network to extract the image features. They then project these features into an embedding space by means of a linear transformation.

Figure 2.39 shows the architecture of the model. The model is trained by minimizing the cosine similarity loss between the encoded image and the encoded recipe. Additionally, they introduced a semantic regularization loss by training a classifier for different food categories. They claim that this shared high-level classification task allows the model to better learn the joint representations of the recipes and images. During inference time, an image is projected into the embedding space and then the recipe with the highest cosine similarity is selected from a corpus of recipes.

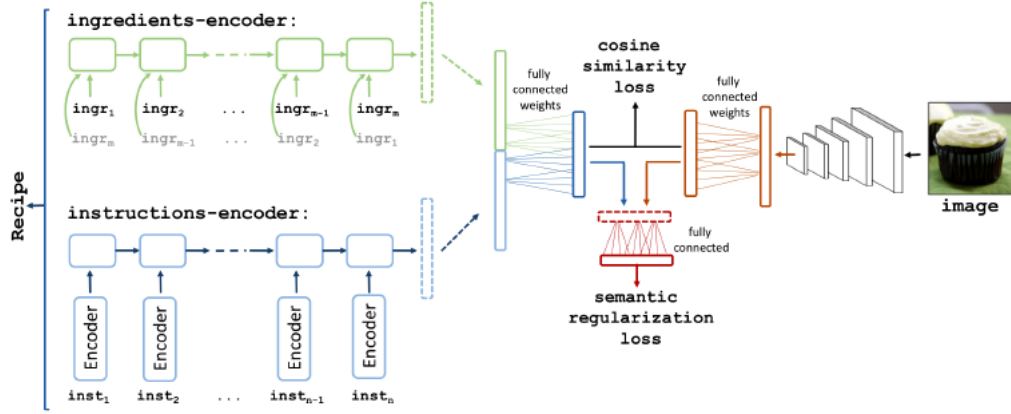


Figure 2.39: The joint neural embedding model for the task of image-to-recipe retrieval proposed by Salvador et al. (2017)

Baddam et al. (2020) presented a different approach where they predict the ingredients of a dish given an input image using a bidirectional LSTM. They then recommend all dishes that match the inferred ingredients.

The approach from Chen et al. (2017a) predicts not only the ingredients, but a whole list of attributes of the dish. These attributes include the ingredients, the cutting as well as the cooking method of a dish. They first feed the input image into a pre-trained VGG network to extract a feature map. The features are then divided into a $m \times m$ grid. They assume that each grid cell contains only one dominant ingredient. Therefore, they predict the attributes for each grid. Finally, they query a recipe based on these attributes. Figure 2.40 illustrates the architecture of the proposed approach.

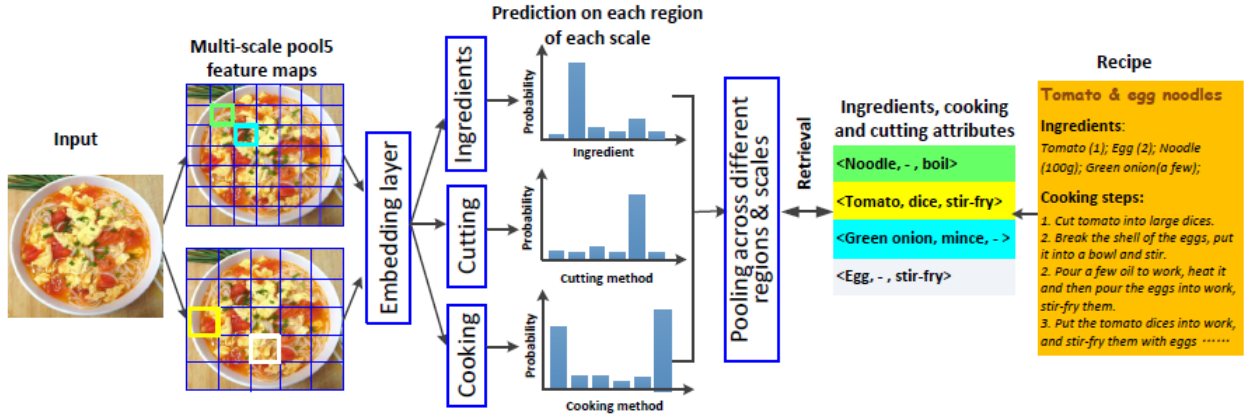


Figure 2.40: The image-to-recipe retrieval approach used by Chen et al. (2017a)

Retrieval-based models have the disadvantage that their performance strongly depends on the number of recipes in the dataset and its diversity. Such systems fail if a suitable recipe for the query image is not available in the dataset.

2.8.2 Recipe Generation

Compared to the image-to-text models that were developed for the task of natural image understanding, generating recipes from images is much more challenging. It requires an understanding of the ingredients as well as all the transformation they went through such as slicing, blending or mixing with other ingredients. Moreover, to detect all ingredients of a dish, the model requires reasoning and prior knowledge. For example, cakes will probably include baking powder and sugar, even though they cannot be detected directly from an image. Furthermore, that a recipe can be actually cooked, it requires a coherent step order. For example, vegetables are usually sliced before they are getting cooked.

Salvador et al. (2018) proposed an inverse-cooking model where they do not simply look up recipes in a database, but generate them by using a language model. More precisely, they generate a title, a list of ingredients and cooking instructions. They compose the generation process into two stages. In the first stage, they predict a list of ingredients from the image. In the second stage, they generate the corresponding title and instructions by applying attention to both the image and the predicted ingredients simultaneously. This form of attention is related to the semantic attention model proposed by You et al. (2016) where they first predict image attributes and then use these attributes to generate the output text. Figure 2.41 illustrates the recipe generation model. The architecture consists of 4 main components: an image encoder, an ingredient decoder and encoder as well as an instruction decoder.

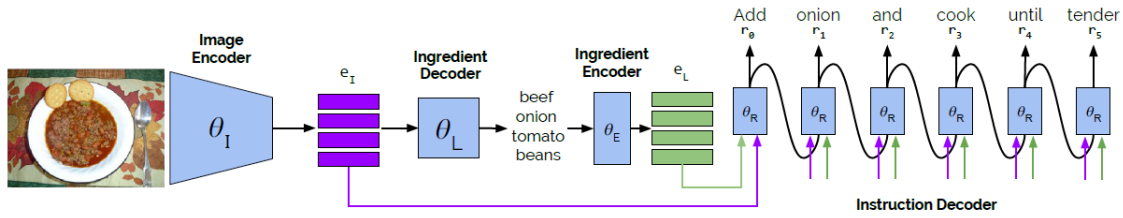


Figure 2.41: An illustration of the inverse cooking model proposed by Salvador et al. (2018)

Image Encoder The authors encode the input image \mathbf{I} into an embedding vector $\mathbf{e}_I \in \mathbb{R}^{L \times d_{\text{model}}}$ by using a pre-trained ResNet-50 encoder, where L are the number of image features and d_{model} is the embedding dimensionality. They use these image features for generating the ingredients and instructions.

Ingredient Decoder The ingredient decoder is a transformer model that is conditioned on the input image embedding \mathbf{e}_I to generate the ingredients. They modelled the ingredient prediction as a set prediction problem by adapting the transformer decoder which they refer as *set transformer*. To remove the order in which ingredients are predicted, they aggregate the outputs across different time-steps by means of a max-pooling operation. Furthermore, to ensure that ingredients are selected without repetition, they mask activations of previously selected ingredients by multiplying them with $-\infty$. The ingredients decoder is trained by minimizing the binary cross entropy between the pooling result of the predicted ingredients and the ground truth. During testing time, they use the ingredients decoder in an autoregressive manner. Since they do not include the end-of-sequence (EOS) token in the binary cross entropy loss, they additionally introduced an EOS loss to learn the stopping criteria. It is defined as the binary cross entropy between the predicted EOS token and the ground truth.

Ingredient Encoder The ingredient encoder consists of an embedding layer followed by a dropout layer to convert the ingredients into the feature space $\mathbf{e}_L \in \mathbb{R}^{M \times d_{\text{model}}}$ which are then used by the instruction decoder. The variable M denotes the number of ingredients.

Instructions Decoder The instructions decoder is used to generate the recipe title as well as the instructions. They regard the title of a recipe as the first instruction of each recipe, separated by a special token. The instruction decoder consists of a transformer decoder which is conditioned on both the encoded image \mathbf{e}_I and the encoded ingredients \mathbf{e}_L . To allow conditioning simultaneously on both embeddings, they concatenate them over the first dimension and then apply attention over the combined embedding $\mathbf{e}_{\text{concat}} \in \mathbb{R}^{(L+M) \times d_{\text{model}}}$.

The model is trained in two stages: first of all, they train the image encoder and ingredient decoder separately. Afterwards, they freeze the weights of the image encoder and train the ingredient encoder and instruction decoder by using the ground truth ingredients as well as the encoded image. To speed up training, they use teacher forcing to train the instructions decoder. In order to diminish the exposure bias problem, they train the ingredients decoder in free running mode, i.e. without teacher forcing. They fixed the maximum number of ingredients to 20. Therefore, the training speed reduction through the lack of teacher-forcing can be neglected.

Another recipe generation model was introduced by Lee et al. (2020). They proposed RecipeGPT, a language model that allows to generate recipe instructions not based on an input image, but on a title and a list of ingredients. The model is a pre-trained GPT-2 language model (Radford et al., 2019) which they fine-tuned on the Recipe1M dataset. They also trained the model in reverse order: based on instructions the model is tasked to generate the according list of ingredients as well as the recipe title.

2.8.3 Metrics

The fact that permuting the ingredients does not change the outcome of the recipe, implies that ingredients can be treated as sets. However, there might be a natural order about how a person writes down the ingredients of a recipe. Nevertheless, Salvador et al. (2018) measures the performance of their ingredient prediction model by means of the IoU and F_1 score from the multi-label classification literature.

Although there are standard metrics for measuring the performance of image-to-text models, there is a high degree of uncertainty about how recipe instructions should be evaluated. Lee et al. (2020) assess the quality by computing the BLEU and ROUGE scores which are often used as metrics for translation and text summarization tasks respectively. Kiddon et al. (2016) additionally measures the METEOR score which considers the word order and synonyms. Two more metrics that have been implemented to measure the quality of instruction texts are explained below.

Ingredients/Instruction Coherence Lee et al. (2020) explain that for the completeness of a recipe, all ingredients should be consistently mentioned in the instructions text. Therefore, they propose an ingredients/instructions coherence score where they calculate the Jaccard similarity between the list of ingredients and the ingredients used in the instructions. To extract all ingredients from the instructions, they select all root nouns and filter out non-ingredients.

Recipe Level Coherence Majumder et al. (2019) claim that n -gram based metrics alone are not sufficient enough to measure the quality of recipe instructions since they can be written in many ways using the same ingredients. They explain that such metrics do not correlate with subjective recipe quality, since generated recipes are more diverse than ground truth recipes. Moreover, they describe that a plausible recipe should possess a coherent step order. To ensure this, they measure the quality of their generated recipes by means of a recipe level coherence score. They divide the full instructions text into a list of individual instructions and encode each using a BERT model. They then train a GRU network that learns the overall instruction ordering structure by minimizing the cosine similarity of the instruction representations presented in the correct and reverse order. Using this GRU network, they then calculate the similarity of a generated instructions text to the forward and backwards ordering of the corresponding ground truth text. This yields a score equal to the difference between the former and latter where a higher score indicates better instructions ordering.

Chapter 3

Setup and Models

This chapter describes the models and the setup that were used to conduct the many experiments in this project. The models were implemented in Python 3.7 with the latest version 2.1 of the TensorFlow framework (Abadi et al., 2016).

3.1 Data Quality Assessment

For the purpose of this project, five cooking websites with recipes written in German were crawled, resulting in a dataset with a total of 0.9M recipes and 1.3M images. Each recipe contains a title, a list of ingredients, an instructions text and optionally one or more images. Table 3.1 shows the different recipe sources with the corresponding number of recipes and images. While the first three websites are popular German cooking portals, the last two are Swiss ones. The recipes from *chefkoch.de* contain 3 images per recipe on average whereas only every third recipe of *kochbar.de* contains an image.

Source	Recipes	Images	Avg. Images per recipe
kochbar.de	531'750	173'423	0.33
chefkoch.de	342'903	1'046'795	3.05
daskochrezept.de	38'498	38'498	1.00
bettybossi.ch	5'218	5'218	1.00
foobie.ch	3'506	3'506	1.00
Total	921'875	1'267'440	1.37

Table 3.1: Different recipe sources with number of recipes

3.1.1 Data Cleaning

Even though the crawled dataset consists of almost 1 million recipes, a lot of them cannot be used because they are of poor quality. Manual inspection revealed that especially community cooking websites contain many recipes that are highly unstructured. To identify such recipes and clean them, the following pipeline was implemented.

Instructions

By manually examining several samples, it has been discovered that some users were using smilies or URLs in the instructions text. These and other special symbols were removed. Moreover, the length of the instructions exhibits high variance: the shortest recipe consists of only one character whereas the longest recipe is 9'282 characters long. The mean recipe contains 760 characters and the median is 661 characters. Figure 3.1 reveals that the distribution is right skewed.

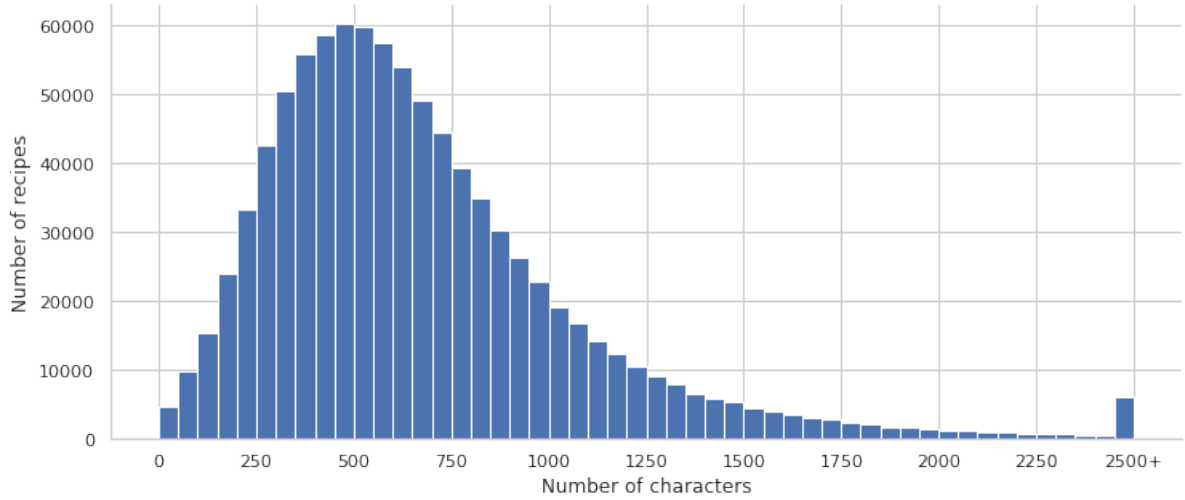


Figure 3.1: The histogram of the instruction lengths in number of characters

To remove outliers, a minimum instruction length of 100 characters has been defined as well as a maximum length of 3'500 characters. The maximum length was determined by means of the elbow plot in figure 3.2, which shows the number of recipes versus the maximum recipe length.

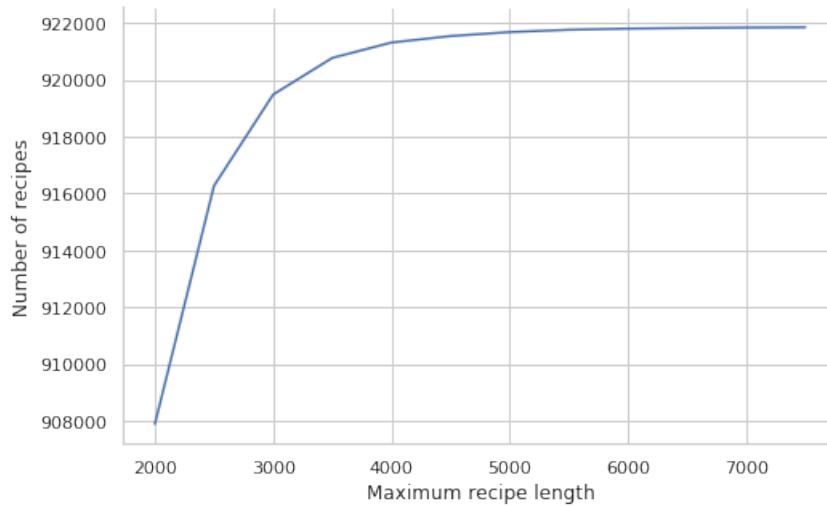
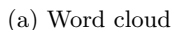


Figure 3.2: Maximum instructions length vs. recipes to drop



(b) Top 10 ingredients

Figure 3.3: Ingredient frequencies before cleaning

Replace special symbols Similar to the cleaning of the instructions, special symbols were either removed or replaced.

Handle Compound Words The German language consists of many compound words such as *"Rindshuftsteak"* or *"Creme Fraiche"*. The first example is a closed compound word and is written as a single word. The latter is an open compound word that is written as two separate words. Identifying open compound words is challenging since classical word tokenizer would treat them as separate words.

Collocations are phrases consisting of more than one word, where the words occur exceptionally often in a corpus. Assuming that collocations are in fact compound words, a list of such words was created before applying the cleaning pipeline. To reduce computation time, only the three datasets of *bettybossi.ch*, *foobie.ch* and *daskochrezept.de* were combined, resulting in a total of 47'222 recipes. All three cooking platforms are professional and it can therefore be assumed that the recipes are of good quality. A corpus of recipe text was generated by concatenating all ingredients and instructions. Based on this corpus, all bigrams were computed. To identify collocations, bigrams are ranked according to the pointwise mutual information (PMI) score introduced by Church and Hanks (1990) and given in equation 3.1. The intuition behind the PMI score is that it measures how much more likely words co-occur than if they were independent. To reduce errors, bigrams that occur less than 10 times were removed before calculating the PMI score. Finally, collocations with a PMI score of at least 10 were kept as a set of reference compound ingredients. Using these reference ingredients, the following rule was implemented during the cleaning process: if an ingredient contains a bigram from the compound words, only this compound word is retained.

$$\text{PMI}(w_1, w_2) = \log \left(\frac{P(w_1, w_2)}{P(w_1) \cdot P(w_2)} \right) \quad (3.1)$$

Splitting multiple ingredients Ingredients that do not consist of compound words are tokenized into multiple words, retaining only nouns. In the best case, only one word is left, which is returned. However, if an ingredient contains more than two nouns it could be that it contains multiple ingredients, such as *"Salz und Pfeffer"* or there was a classification error by the part-of-speech (POS) tagger.

To minimize errors in these steps, a reference ingredient set was calculated prior to the cleaning process. Manual inspection showed that the dataset from *bettybossi.ch* is the most structured one. This is mainly because it is not a community cooking website: only Betty-Bossi employees can upload new recipes, so it can be assumed that they maintain a standardized list of ingredients. Thus, the list of 3'606 unique ingredients is much smaller than from the other websites. These ingredients were taken as reference after removing all non-noun words using a POS tagger. Since it is a Swiss cooking website, there are many Helvetism words such as *Rüebli* instead of the German word *Karotten*. Therefore, the reference set was manually extended and revised, leading to a set of 2'411 unique ingredients. If an ingredient contains more than two nouns, the set of reference ingredients comes into play: only nouns that are contained in this set are kept.

Merge singular and plural For each of the ingredients, the number of occurrences in the dataset are calculated as well as their stem. Ingredients that share the same stem are merged together by taking the word that occurs the most frequently in the dataset. This allowed singular and plural words to be merged.

Remove typos Since the majority of the recipes were written by amateurs, many typos can be found. To identify such typos, ingredients that have a Levenshtein distance of one are merged together. To avoid errors such as the conversion of the word *"Eis"* into *"Reis"*, only ingredients longer than 3 characters were considered. If an ingredient is longer than 16 characters, the maximum Levenshtein distance is increased to 2.

Merging synonyms To identify synonyms, FastText word-embeddings were retrained on the ingredients and instructions of the whole dataset. Using these embeddings, ingredients that have a cosine similarity of at least 0.9 are considered as synonyms and thus merged together.

Remove infrequent Ingredients that appear only a few times are hard to predict. The plots in figure 3.4 show how the number of recipes (a) and ingredients (b) change by increasing the minimum occurrence threshold. It is a trade-off between losing recipes and reducing complexity by decreasing the number of ingredients. Based on these plots, recipes that contain ingredients that appear less than 150 times were removed.

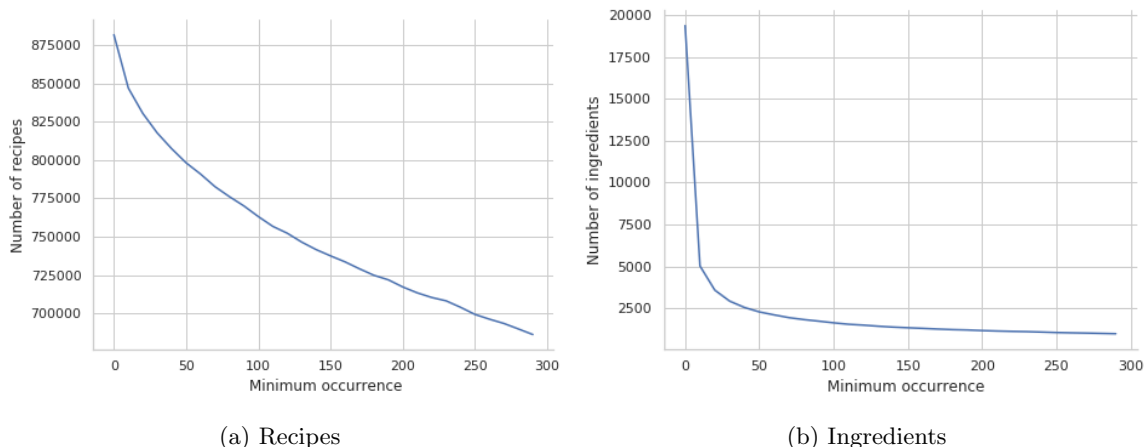


Figure 3.4: Elbow plot of the minimum ingredient occurrence

After cleaning the ingredients, the number of ingredients per recipe was analysed: there are recipes that contain no ingredients and some with over 40. 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 (as depicted in figure 3.5), a maximum length of 19 ingredients was defined. Recipes that contain less than 2 ingredients were considered useless and thus also removed.

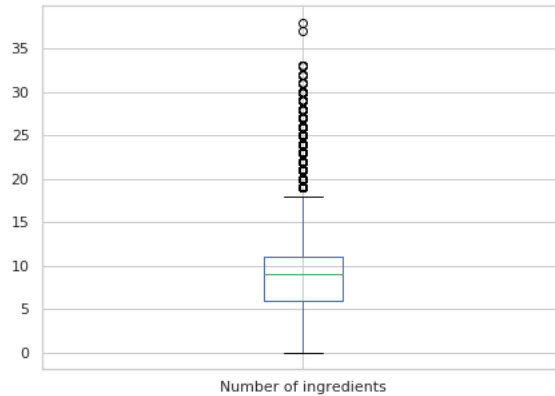


Figure 3.5: Boxplot of the number of ingredients

Altogether, the number of unique ingredients could be reduced from around 1 million to 1'332 using these cleaning techniques.

Images

To use a recipe for the training of an image-to-text model, at least one image of the dish with sufficient resolution is required. Therefore, all images with a resolution lower than 190×190 were removed. Since the dataset can also be used for other tasks that do not require images, recipes without images were retained at this stage.

3.1.2 Data Analysis

The raw dataset with 921'875 recipes has been transformed into a reduced dataset of 722'823 during the cleaning phase. Further analysis of the cleaned data was conducted to get a deeper understanding of the data. The most important results are presented in the following subsection.

Recipe Titles

There are in total 538'957 unique recipe titles, which means that some titles appear multiple times. The title *Nudelsalat* is the leader and occurs in 458 different recipes. Table 3.2 shows the top 5 recipe titles.

Title	Occurrences
Nudelsalat	458
Tiramisu	374
Kartoffelsalat	339
Apfelkuchen	333
Kartoffelsuppe	315

Table 3.2: Recipe titles with its number of occurrences

The summary statistics in table 3.3 show that there is a high variance in the length of the recipe titles. While the shortest titles consist of only one word, the longest ones consist of 26 words.

	Characters	Words
Mean	25.49	2.92
Median	23	3
Standard Deviation	12.01	1.86
Min	1	1
Max	157	26

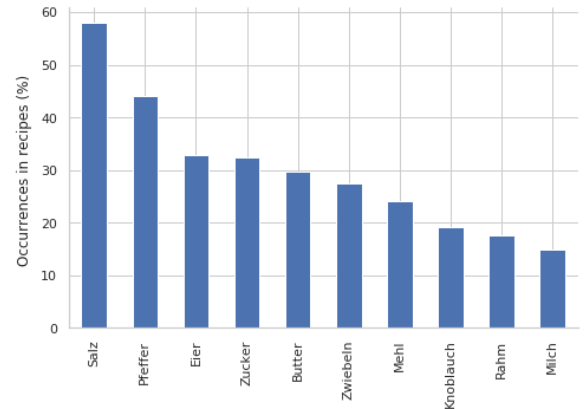
Table 3.3: Statistics of the recipe titles

Ingredients

Figure 3.6 shows the ingredient frequencies after the cleaning process. Again, *Salz* is the most frequent ingredient and occurs in about 60% of the recipes.



(a) Word cloud



(b) Top 10 ingredients

Figure 3.6: Ingredient frequencies after cleaning

The cumulative occurrences of the ingredients in percentage are depicted in figure 3.7. It clearly indicates that they are imbalanced: only 250 ingredients account for 90% of all occurrences. This fact has to be kept in mind when evaluating the implemented models.

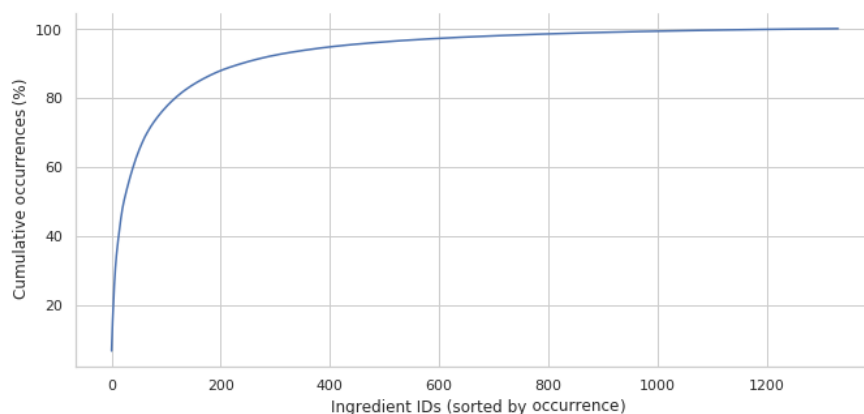


Figure 3.7: Cumulative sums of the ingredient occurrences in percentage

Computing frequent itemsets using the Apriori algorithm (Agrawal and Srikant, 1994) reveals which ingredient often co-occur together. Table 3.4 shows that both spices *Salz* and *Pfeffer* occur in 40% of all recipes together. Given that *Pfeffer* occurs in a total of 44% of all recipes means that it occurs only in 4% of all recipes without the ingredient *Salz*. Therefore, when predicting *Pfeffer* it is highly likely that the model needs to predict *Salz* as well.

Itemsets	Support (%)
Salz, Pfeffer	40.15
Zwiebeln, Salz	21.52
Zwiebeln, Pfeffer	20.03
Salz, Eier	18.20
Mehl, Eier	17.76

Table 3.4: Frequent itemsets of ingredients

Instructions

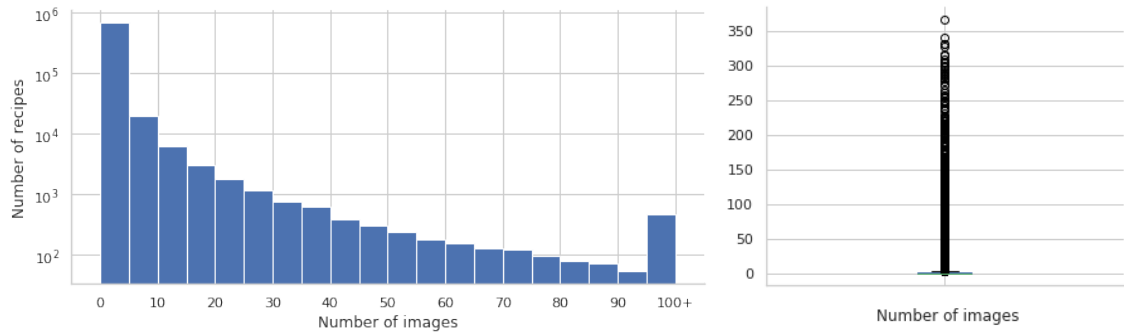
The instructions of a recipe contain on average 111 words and 10 sentences. The word cloud in figure 3.8 shows the most common words and collocations from the instructions text. Again, the most common ingredients *Salz* and *Pfeffer* occur in the plot.



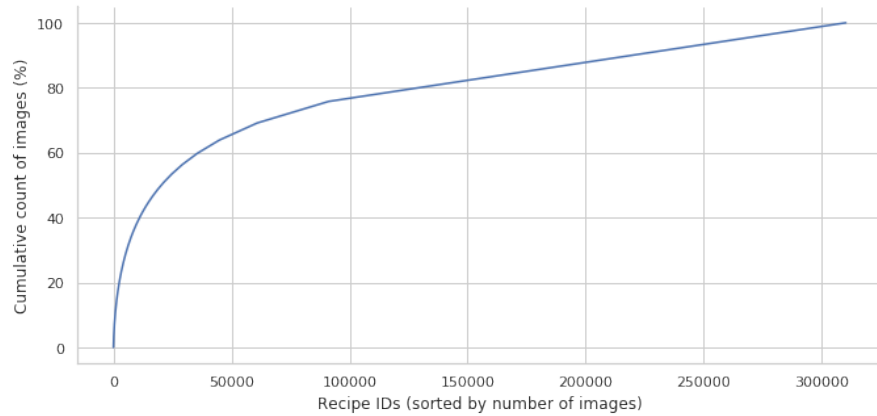
Figure 3.8: A word cloud of the instruction texts that contains the most frequent words and collocations

Images

The cleaned dataset contains a total of 903'984 images, which means that some recipes have multiple images. However, only 310'136 recipes have at least one image. Examining the number of images per recipe reveals that there are many outliers with an exceptionally high number of images. The histogram for the number of images depicted in figure 3.9a shows that the distribution is heavily right skewed. The box plot in figure 3.9b further indicates that there are recipes without and with more than 300 images. Moreover, figure 3.9c shows the cumulative sum of the number of images: almost 25% of all images belong to only 1% of the recipes.



(a) Histogram of the number of images with log-scaled y -axis (b) Box plot of the number of images



(c) Cumulative sum of the number of images

Figure 3.9: Number of images per recipe

Table 3.5 displays the recipes with the highest number of images. It can be assumed that these are popular recipes from community cooking platforms that users have tried out and uploaded pictures of their results.

Title	Images
Barbie-Torte	366
Zitronentorte	340
Zimtrolle-Kuchen	331
Lasagne	330
Schneller Flammkuchen	327

Table 3.5: Recipes with the highest number of images

Figure 3.10 contains 100 images of the recipe *Lasagne* which contains 330 images in total. By examining these pictures it can be noted that there are images from several stages of the recipe preparation process: before baking the lasagne and after. This makes it rather difficult for the model to learn all variations.



Figure 3.10: Images of the recipe *Lasagne* from the dataset

3.2 Preprocessing

In order to train machine learning models, the data needs to be preprocessed. The first step was to remove all recipes that do not contain at least one image. Based on the findings of the data quality assessment, the following preprocessing steps were executed afterwards.

3.2.1 Dataset Splitting

The full dataset is split into a training, validation and test set. According to the best practices by 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 60% of the whole dataset.

Validation Set The validation set is used to tune the hyperparameters and select the best model. It contains 20% of the data.

Test Set The test set contains 20% data 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 are selected and trained. If some data is leaked, it may happen that the evaluated performance is optimistically biased.

Split	Recipes	Images
Training	186'081	544'552
Validation	62'028	178'214
Test	62'027	181'218
Total	310'136	903'984

Table 3.6: Number of samples in training, validation and test set

3.2.2 Input Pipeline

An input pipeline was developed that loads the training examples. Each example is a tuple $(\mathbf{I}, \mathbf{y}^{(\text{ingr})}, \mathbf{y}^{(\text{instr})})$, where \mathbf{I} denotes the image, $\mathbf{y}^{(\text{ingr})}$ the list of ingredients and $\mathbf{y}^{(\text{instr})}$ the instructions text and the recipe title.

Neural networks have the constraint that the shape of each mini-batch must be identical. For the images, this can be achieved by resizing them to a predefined shape. The ingredients are padded to the maximum length of each batch. For the instructions, however, this would be very inefficient, since they exhibit a high variance in their lengths. On the one hand, there are instructions that contain only 100 characters and on the other hand, there are instructions with 3'500 characters. Also, a small batch size would be required to avoid memory issues. To allow the usage of a higher batch size and thus increase the effectiveness of training, the maximum instruction length could be decreased. However, even more recipes would be lost. Therefore, the bucketing algorithm that has been successfully used by Khomenko et al. (2017) was applied for the instructions. It creates buckets for training examples that contain instructions of similar length. This way, the batch size is not fixed but varies according to the current samples. In order to fully utilize the provided hardware, the maximum tokens per batch was set to 8'000. With the used training set, this results in a batch size of around 52 samples on average.

Images

The cleaned dataset contains on average about 3 images per recipe. During training time, for each of the recipes an image is randomly selected from the available images. This allows the model to learn from more examples.

Too little training data leads to overfitting and thus poor generalization. Wong et al. (2016) successfully showed that data augmentation can be used to reduce overfitting by artificially generating additional samples from the original ones. Inspired by these findings, recipe images are randomly flipped horizontally. This way more training data is available and the model learns to be more robust. Because the model requires an input image of fixed size, the image is resized in a way that the shortest edge satisfies the size constraints. Then, the image is randomly cropped. These augmentation techniques can either be executed before training or during training time by augmenting the currently loaded batch. However, performing the augmentation before training has the drawback that it consumes additional storage space, thus the latter approach was selected. Additionally, the hue, saturation, brightness and contrast values of the images were randomly adjusted. However, experiments revealed that this could not improve the performance. Figure 3.11 illustrates some augmented training samples where the first image per row shows the original image, the second the centrally cropped image, and the last two with random cropping and flipping.



Figure 3.11: Examples of data augmentation

Ingredients

The ingredients are converted into tokens by using a standard word-encoder with a fixed-sized vocabulary. Additionally, a start-, end- and padding-symbol is added. Lee et al. (2020) explain that ingredients can be treated as sets and thus the order should not matter. Therefore, they shuffle the ingredients during training to ensure that the model learns to generate recipes, no matter on what order its ingredients were written down. To investigate whether shuffling improves the result, it was implemented.

Instructions and Recipe Title

Salvador et al. (2018) treats the recipe title as the first instruction of a recipe. As a result, the model requires fewer parameters, as no additional decoder for generating the title is required. This multi-field preprocessing was adapted in this project by concatenating the recipe title and the instructions text separated by a special end-of-title token.

The recipe instructions and titles contain 531'809 unique tokens, which requires a huge amount of memory if each token is one-hot encoded. Moreover, 97.5% of the words appear less than 100 times. Therefore, a subword-unit encoder was used that performs byte pair encoding (BPE) tokenization. It allows to define the number of unique tokens in the vocabulary. Based on experiments on the validation set, a vocabulary size of $2^{14} = 16'384$ unique tokens was selected. Further reducing the vocabulary size can lead to longer training time, as the sequences will increase in length. Figure 3.12 illustrates this multi-field preprocessing technique followed by a subword-unit encoder.

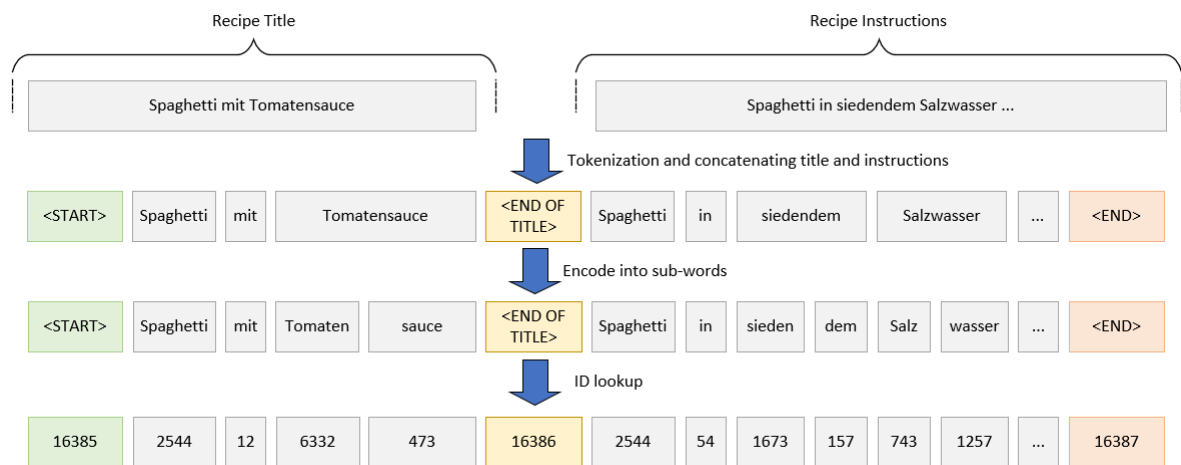


Figure 3.12: Tokenization of the recipe title and instructions text

3.3 Metrics

The goal of the project is to implement a model that is able to generate a recipe consisting of a title, a list of ingredients and an instruction text based on an input image. To measure the performance of each of those subtasks, different metrics are required.

3.3.1 Recipe Title

The title should give a brief overview of the recipe. Data analysis showed that the median of the titles consists of 3 words. Consequently, it makes no sense to measure the quality with a score that considers n -grams of higher order than 3. Moreover, the order in which the words of a title occur is not as important as in the instructions text. Thus, to measure the quality of the title, the BLEU score with $N = 1$ (Bl_1) was calculated, considering only single words.

3.3.2 Ingredients

Followed by Salvador et al. (2018), the ingredient outputs were treated as sets. Therefore, example-based metrics from image-to-set prediction models were measured such as the IoU, precision (Pr), recall (Re) and F_1 score.

3.3.3 Instructions

Recipe instructions are long and coherent texts describing how the dishes are cooked. In the following, all standard image-to-text metrics are briefly reviewed to determine if they are applicable for the project.

BLEU Score (Bl_N) The BLEU score measures the precision computed by n -gram overlaps and punishes too short outputs. It allows to use multiple reference outputs as ground truth values. Since the same recipe can be formulated in different ways, the ground truth recipes were grouped by name. Analysis showed that there are for example 458 recipes for the dish *Nudelsalat*. So to evaluate such a recipe instruction, all those references are used to compute the BLEU score. The score was computed with the parameters $N = 1$, $N = 2$ and $N = 4$ with uniform weights.

ROUGE Score (Ro_n) The ROUGE score is very similar to the BLEU score, however, it also measures the recall. For evaluating the instructions, the ROUGE F_1 score was computed with the parameter $n = 4$.

METEOR Score The advantage of the METEOR over other n -gram based metrics is that it considers synonyms. To the best of the authors' knowledge, there is no open-source database of synonyms for the German language such as WordNet for English. Therefore, the METEOR score cannot be used to evaluate the system trained in this project.

SPICE Score The SPICE score has the advantage over the other scores that it considers semantically similar sentences. Nevertheless, like the METEOR score, it requires a database of synonyms. Thus, it is excluded for further analysis in this project.

CIDEr Score (Ci_n) The CIDEr score stems the instructions and measures the cosine similarity between n -gram TF-IDF representations. The intuition behind this score is to penalize common n -grams. Analysis of the instructions has shown that there are frequent collocations such as "*Scheiben schneiden*" or "*Minuten backen*". The CIDEr score penalizes such collocations, therefore, the score has to be taken with care. Nevertheless, it has been calculated to evaluate the instructions text. More specifically, the CIDEr-D score is used since it is one of the main metrics for image captioning challenges such as the annual COCO challenge (Chen et al., 2015).

WMD Score The advantage of the WMD score over the other metrics is that it considers semantically similar words by using word embeddings. However, as the computation is very time consuming, it was not further considered.

Ingredients/Instructions Coherence (I/I) In order to measure whether the predicted ingredients actually occur in the instructions text, a metric similar to that proposed by Lee et al. (2020) was implemented. During the data cleaning phase, a list of synonyms was generated for each ingredient. Before computing this metric, all ingredients and instructions are stemmed. For each ingredient, there is a match if the stemmed ingredient or a synonym occurs at least once in the instructions text. The number of matches is then divided by the total number of predicted ingredients, which results in a precision-based score. To compute the score for the whole corpus, the mean is calculated.

Recipe Level Coherence The recipe level coherence score introduced by Majumder et al. (2019) measures the correct ordering of the instructions. The computation of the metric requires access to a pre-trained German BERT model and the training of an additional scoring network. Due to the complexity of this metric, it was not considered in this project.

Using the sub-word tokenized instructions to compute the n -gram based scores, such as BLEU or ROUGE, would result in too optimistic metrics. This is because words constructed from multiple sub-words would count as n -gram matches. To compute the metrics, the instructions were therefore encoded into words by means of a standard word tokenizer.

3.4 Baseline Model

For better interpretability of the metrics, a baseline model was implemented. As noticed during the data quality assessment phase, the ingredients are highly imbalanced. Moreover, the median number of ingredients per recipe is 9. The top 9 ingredients account already for almost 33% of the total occurrences. Furthermore, 91% of the recipes contain at least one of these ingredients.

Based on these statistics, the baseline model illustrated in figure 3.13 was implemented that returns the top 9 ingredients from the training set. The instructions and recipe titles are randomly selected from all recipes of the training set that contain at least one of these ingredients.

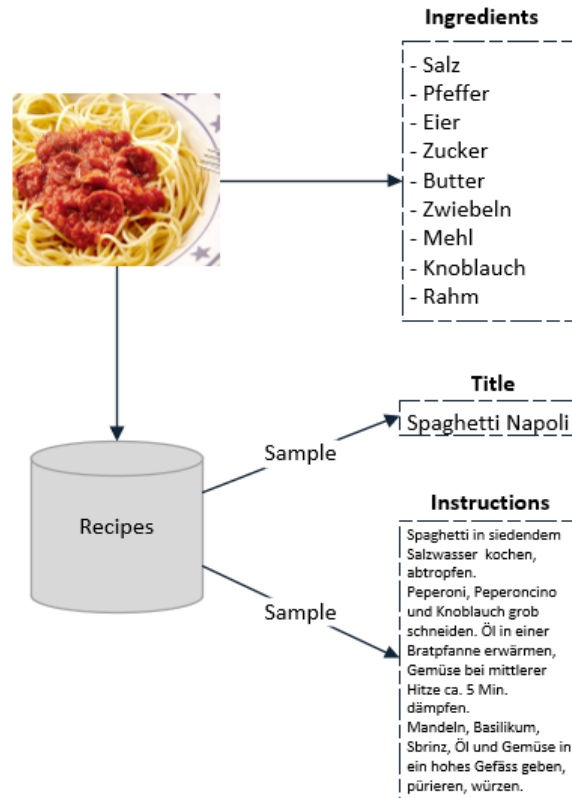


Figure 3.13: An illustration of the baseline model

3.5 Inverse Cooking Model

Inspired by the work of Salvador et al. (2018), an inverse cooking model was implemented that first generates a list of ingredients based on an image and then generates an instruction text conditioned on both the image features and the ingredients. Figure 3.14 illustrates the architecture of the inverse cooking model. It consists of an image encoder, an ingredients decoder and an instructions decoder.

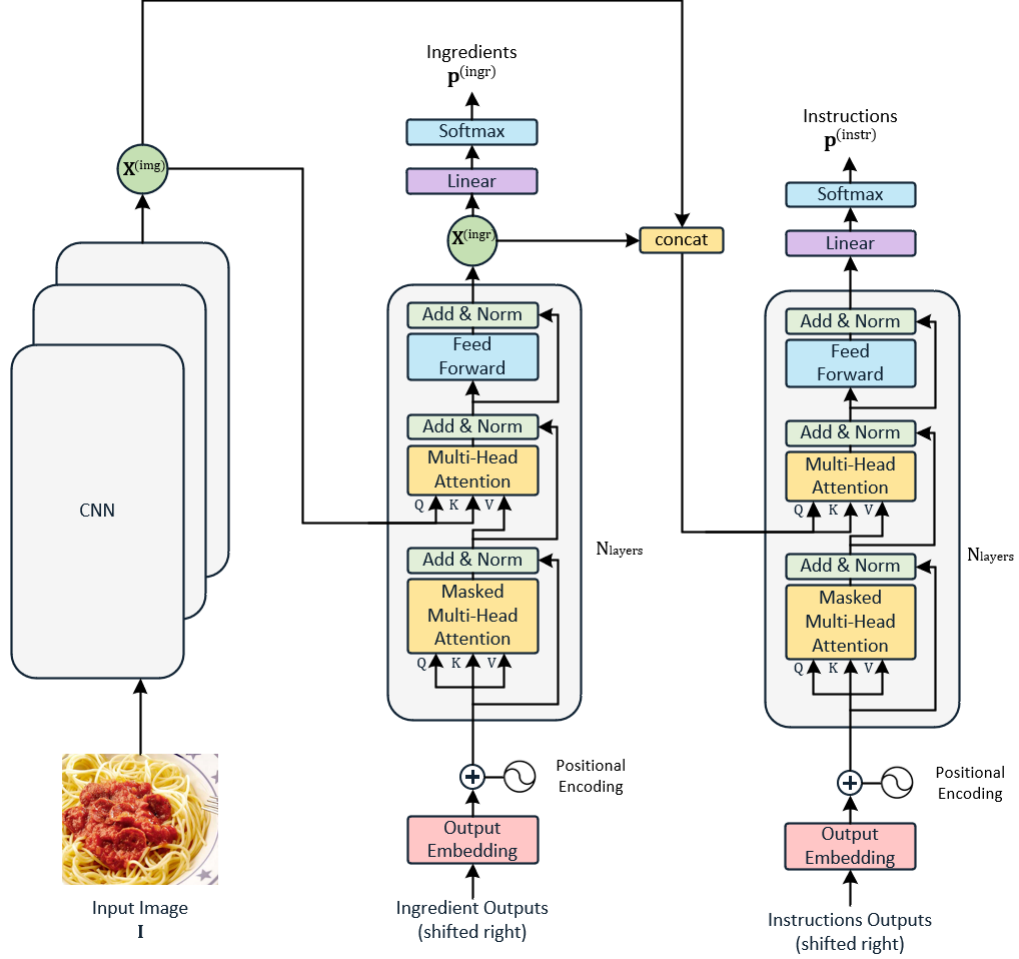


Figure 3.14: An overview of the proposed recipe generation model

3.5.1 Image Encoder

The image encoder maps an input image $I \in \mathbb{R}^{w \times h \times 3}$ into a feature representation $X^{(img)} \in \mathbb{R}^{L \times d_{model}}$ where L denotes the number of image features and d_{model} the embedding dimension. Four CNN architectures were compared: VGG-16, VGG-19, ResNet-50V2 and Inception-V3. To employ them for feature extraction, the last fully connected layer of the original networks used for classification was removed. Transfer learning was applied to all types of networks, using pre-trained weights from the ImageNet challenge. All layers were unlocked, allowing the model to optimize the full network. Springenberg et al. (2018) proposed to apply self-attention to the encoded image features. Based on these findings, self-attention was implemented for the image encoder. However, as presented in the subsequent chapter, the performance of the model could not be improved by this technique. Therefore, no such layers were used for the final configuration. Furthermore, the VGG-16 feature extractor showed to generate more accurate features. The full configuration of the image encoder is given in table 3.7.

Setting	Value
Input image width, w	224
Input image height, h	224
Number of image features, L	49
Features dimension, d_{model}	512
Feature extractor	VGG-16
Number of self-attention layers	0
Number of attention heads	8

Table 3.7: Setup of the image encoder

3.5.2 Ingredients Decoder

Even though the performance of the ingredients decoder is evaluated by using multi-label classification metrics, the generation of the ingredients is modelled with an autoregressive model. It is a transformer decoder network that is conditioned on the image features $\mathbf{X}^{(\text{img})}$ to produce a feature vector $\mathbf{X}^{(\text{ingr})} \in \mathbb{R}^{M \times d_{\text{model}}}$, with M denoting the number of ingredients. The outputs of the decoder are then passed into a linear layer followed by a softmax function to produce the predictions $\hat{\mathbf{y}}^{(\text{ingr})} \in \mathbb{R}^{M \times \text{vocab}^{(\text{ingr})}}$, where $\text{vocab}^{(\text{ingr})}$ denotes the vocabulary size. Table 3.8 lists the settings of the ingredients decoder that were tuned based on experiments on the validation set.

Setting	Value
Features dimension, d_{model}	512
Number of layers	3
Number of attention heads	8
Dimension feed forward	2048
Hidden activation	ReLU
Linear layer units, $\text{vocab}^{(\text{ingr})}$	1'335

Table 3.8: Setup of ingredients decoder

Loss Function

As described in section 2.4.2, using an autoregressive model for a multi-label classification task introduces an intrinsic label ordering during training. To diminish this effect, the ingredients were not positionally encoded. The ingredients decoder was optimized using the categorical cross entropy loss which is commonly used for transformer models. Moreover, since the output can be treated as a set, several losses used for image-to-set models were evaluated. To convert the predictions into a binary vector, max-pooling over the number of ingredients M was applied. As depicted in figure 3.15, this procedure results in a prediction vector $\hat{\mathbf{z}}^{(\text{ingr})} \in \mathbb{R}^{\text{vocab}^{(\text{ingr})}}$. The ground truth values were one-hot encoded and then pooled over the number of ingredients to obtain a vector $\mathbf{z}^{(\text{ingr})} \in \{0, 1\}^{\text{vocab}^{(\text{ingr})}}$.

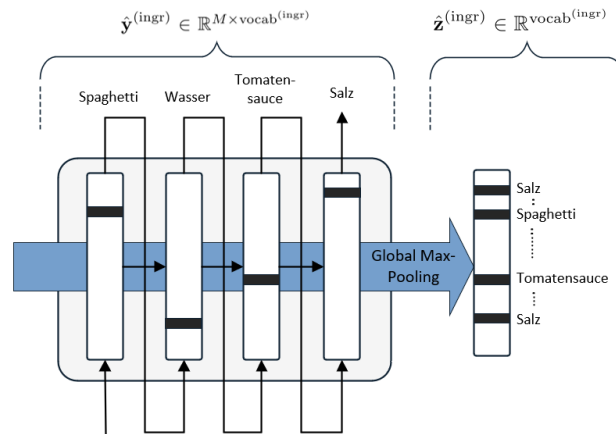


Figure 3.15: Global max-pooling applied to the generated ingredients

The evaluated loss functions are presented below. For simplicity, the superscript (**ingr**) has been removed.

Categorical Cross Entropy The categorical cross entropy considers the order of the ingredients and is the standard loss function when using autoregressive models for text-generation. It is computed as follows:

$$\mathcal{L}_{\text{CCE}} = - \sum_{t=1}^M \sum_{i=0}^{\text{vocab}} y_{t,i} \cdot \log(\hat{y}_{t,i}) \quad (3.2)$$

Soft IoU Loss The soft-IoU loss was first introduced in the context of semantic segmentation (Rahman and Wang, 2016) and then successfully applied for training multi-label classification models. It is an approximation of the IoU and computed as follows:

$$\mathcal{L}_{\text{sIOU}} = 1 - \frac{\sum_{i=1}^{\text{vocab}} \hat{z}_i z_i}{\sum_{i=1}^{\text{vocab}} \hat{z}_i + z_i - \hat{z}_i z_i} \quad (3.3)$$

Binary Cross Entropy When converting the prediction and ground truth values into binary vectors, the binary cross entropy loss can be calculated as given in equation 3.4. It has the advantage over the categorical cross entropy that it does not consider the order of the ingredients.

$$\mathcal{L}_{\text{BCE}} = - \sum_{i=1}^{\text{vocab}} (z_i \cdot \log(\hat{z}_i) + (1 - z_i) \cdot \log(1 - \hat{z}_i)) \quad (3.4)$$

End-of-Sequence Loss As previously described, the vocabulary has been extended by a special EOS token, indicating the end of a sequence. When the categorical cross entropy is used, this special token allows to learn the cardinality of the sequence. However, if for example only the binary cross entropy loss is used, an additional loss is required to learn the stopping criteria, because binary cross entropy does not consider the word order. Salvador et al. (2018) proposed an end-of-sequence loss which calculates the binary cross entropy between the predicted EOS probability at all time steps and the ground truth.

Each of these losses is weighted by a parameter λ , leading to the final definition of the ingredients loss:

$$\mathcal{L}_{\text{ingr}} = \lambda_{\text{CCE}} \mathcal{L}_{\text{CCE}} + \lambda_{\text{sIOU}} \mathcal{L}_{\text{sIOU}} + \lambda_{\text{BCE}} \mathcal{L}_{\text{BCE}} + \lambda_{\text{EOS}} \mathcal{L}_{\text{EOS}} \quad (3.5)$$

Salvador et al. (2018) used only the binary cross entropy in combination with the end-of-sequence loss. However, using this approach led to an exposure bias problem where the model always predicted the same ingredients during testing time. Running several experiments resulted in the configuration given in table 3.9.

Setting	Value
λ_{CCE}	1
λ_{sIOU}	10^{-3}
λ_{BCE}	0
λ_{EOS}	0

Table 3.9: Setup of ingredients decoder loss

3.5.3 Instructions Decoder

The instructions decoder is a transformer decoder network that is conditioned on both the embedded image $\mathbf{X}^{(\text{img})}$ and the ingredients features $\mathbf{X}^{(\text{ingr})}$. This is done by concatenating both features over the first dimension which produces a feature vector $\mathbf{X} \in \mathbb{R}^{(L+M) \times d_{\text{model}}}$. The output of the transformer decoder is again fed into a linear layer followed by softmax to produce a probability distribution over the vocabulary of instructions $\mathbf{p}^{(\text{instr})}$. The final settings are visible in table 3.10.

Setting	Value
Features dimension d_{model}	512
Number of layers	6
Number of attention heads	8
Dimension feed forward	2048
Hidden activation	ReLU
Linear layer units, vocab ^(instr)	16'384

Table 3.10: Setup of the instructions decoder

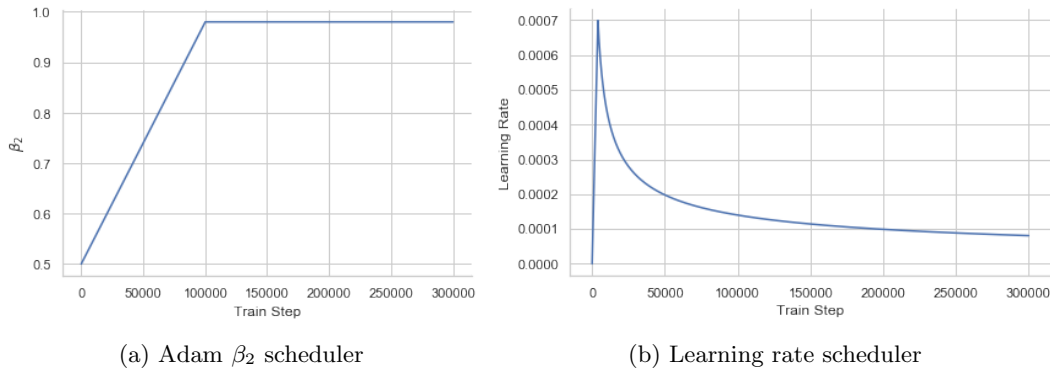
Loss Function

To train the instructions decoder, the standard categorical cross entropy loss is used. Since it is a sequence prediction task where the order of the output matters, no other loss functions were tried out.

3.5.4 Training

Salvador et al. (2018) suggested to train the ingredients decoder without teacher forcing. This was tried out, however, the experiments led to unsatisfactory results. Therefore, both the ingredient and instructions decoder were trained using the teacher forcing technique. The model was trained with the Adam optimizer by using the default decay rate β_1 of 0.9. The parameter β_2 is linearly increased from 0.5 to a maximum of 0.98 over a total of 100'000 steps. Experiments on the validation set showed that this scheduler resulted in faster training time because the effective learning is increased at the beginning of the training. In addition, Vaswani et al. (2017) suggested to use a learning rate scheduler that linearly increases the learning rate during a warm-up period and then decays it over time as given in equation 3.6. This learning rate scheduler was implemented with a total of 4'000 warm-up steps. Both implemented schedulers are plotted in figure 3.16.

$$\text{learning_rate} = d_{\text{model}}^{-0.5} \cdot \min(\text{step_num}^{-0.5}, \text{step_num} \cdot \text{warmup_steps}^{-1.5}) \quad (3.6)$$

Figure 3.16: The implemented β_2 and learning rate scheduler

The tuned optimizer settings are visible in table 3.11.

Setting	Value
Optimizer	Adam
Decay rate β_1	0.9
Max decay rate β_2	0.5 to 0.98 (Scheduler)
Learning rate	Scheduler
Number of epochs	100

Table 3.11: Optimizer configuration of the recipe model

Given a training example $(\mathbf{I}, \mathbf{y}^{(\text{ingr})}, \mathbf{y}^{(\text{instr})})$, the model is trained using a joint-learning approach where the loss is computed as a linear combination between the ingredient $\mathcal{L}_{\text{ingr}}$ and the instructions loss $\mathcal{L}_{\text{instr}}$:

$$\mathcal{L} = \lambda_{\text{ingr}} \mathcal{L}_{\text{ingr}} + \lambda_{\text{instr}} \mathcal{L}_{\text{instr}} \quad (3.7)$$

where λ_{ingr} and λ_{instr} denote the factors for the two losses. Both factors were set to 1.0.

On the contrary, Salvador et al. (2018) train their model in two stages: the ingredients decoder is trained in the first stage and those weights are frozen. In the second stage, the instructions decoder is trained by using the ground truth ingredients. To encode those ingredients into the feature space, use an additional embedding layer which they refer as ingredients encoder. Such a two-stage approach has the disadvantage that it requires more time since two models need to be trained sequentially. Tests revealed that the performance could not be increased when using the two-stage approach.

Regularization

To prevent the model from overfitting, dropout with a rate of 0.2 is applied to the output of each sub-layer. Additionally, label smoothing is used for both the ingredients and instructions decoder with the parameter $\epsilon_{\text{ls}} = 0.1$. Label smoothing was first introduced by Szegedy et al. (2015) in the context of image classification and modifies the ground truth labels so that the model does not minimize the loss in respect to hard targets but to soft targets. It is computed as follows:

$$y_k^{\text{ls}} = y_k(1 - \epsilon_{\text{ls}}) + \frac{\epsilon_{\text{ls}}}{K} \quad (3.8)$$

where K denotes the number of classes and ϵ_{ls} the smoothing factor.

3.5.5 Inference

To produce the outputs $\hat{\mathbf{y}}^{(\text{ingr})}$ and $\hat{\mathbf{y}}^{(\text{instr})}$ at inference time, the input image \mathbf{I} is passed into the image encoder to extract the image features $\mathbf{X}^{(\text{img})}$. Then, the ingredients $\hat{\mathbf{y}}^{(\text{ingr})}$ are predicted in an autoregressive manner: at each time step t , the previously predicted ingredients and the image features are passed to the ingredients decoder to obtain the vocabulary distribution $\mathbf{p}_t^{(\text{ingr})}$. To sample from this distribution, the beam search algorithm was implemented according to the formula from Wu et al. (2016) (see section 2.7.8) with the proposed parameter $\alpha = 0.6$. Additionally, to ensure that ingredients are selected without repetition, activations of previously selected ingredients are masked by multiplying them with $-\infty$. This is repeated until the end-token is predicted or the maximum number of ingredients is exceeded. The predicted ingredients are then passed again to the ingredients decoder to obtain their feature representation $\mathbf{X}^{(\text{ingr})}$. The instructions $\hat{\mathbf{y}}^{(\text{instr})}$ are predicted in the same way as the ingredients, however, the decoder attends additionally to the ingredient features $\mathbf{X}^{(\text{ingr})}$ and there is no masking of previously selected words.

Chapter 4

Results

This chapter presents the results of the trained inverse cooking models and compares them according to their performance. Finally, a model configuration is selected for further evaluation.

4.1 Model Comparison

The following section compares different model configurations based on their performance measured on the validation set and their number of parameters. The model configurations listed below were evaluated.

Baseline The baseline model that returns the top 9 ingredients and samples recipe titles and instructions for recipes containing these ingredients.

- Ⓐ **VGG-16-NoIngr** This configuration uses a pre-trained VGG network with 16 layers to extract the image features. Unlike all other configurations, the instructions decoder is only conditioned on the image features.
- Ⓑ **VGG-16** This configuration uses again a pre-trained VGG network with 16 layers as the backbone of the image encoder. In comparison to Ⓐ, the image and ingredient features are concatenated before passing them to the instructions decoder. This way the model can apply attention to both features simultaneously. All subsequent configurations use this procedure.
- Ⓒ **VGG-19** In this configuration, the VGG-16 was replaced by the slightly deeper network VGG-19 for the extraction of image features.
- Ⓓ **ResNet-50V2** This configuration uses a pre-trained ResNet-50V2 feature extractor as the backbone of the image encoder.
- Ⓔ **Inception-V3** This configuration uses the Inception-V3 architecture consisting of 48 layers, also known as GoogLeNet, for extracting the image features. In contrast to the other evaluated image encoders, this configuration produces 25 features instead of 49.
- Ⓕ **VGG-16-SelfAttn** Springenberg et al. (2018) reports increased performance of their image captioning system when applying self-attention to the image features. Therefore, this configuration extends the VGG-16 image encoder with 2 self-attention layers before passing the features to the ingredients and instructions decoder.
- Ⓖ **VGG-16-GTIngr** This configuration uses again the VGG-16 network to extract the image features. However, it uses the ground truth ingredients to train the instructions decoder as proposed by Salvador et al. (2018). The ground truth ingredients are passed to an additional embedding layer to obtain a feature representation required for the instructions decoder.

4.1.1 Metrics

The model configurations were compared by means of the metrics calculated on the validation set. All models were evaluated with beam search using beam size 1, i.e. greedy search. The results are visible in table 4.1, where the highest values per metric are highlighted.

Model	Ingredients				Title	Instructions					
	IoU	Pr	Re	F ₁	Bl ₁	Bl ₁	Bl ₂	Bl ₄	Ro ₂	Ci ₄	I/I
Baseline	20.28	33.11	33.71	32.18	3.14	27.21	11.40	1.97	4.07	0.62	25.26
Ⓐ VGG-16-NoIngr	26.60	40.26	41.60	39.15	7.95	31.03	17.50	6.10	9.85	3.68	34.35
Ⓑ VGG-16	26.15	39.05	41.69	38.62	7.12	32.28	17.97	6.08	9.54	3.28	61.71
Ⓒ VGG-19	24.13	37.26	38.02	36.03	4.51	33.06	17.82	5.63	8.66	2.51	71.35
Ⓓ ResNet-50V2	19.87	31.65	32.23	30.55	1.13	29.54	15.05	3.90	7.05	1.29	73.67
Ⓔ Inception-V3	25.17	38.92	39.14	37.35	6.85	32.92	18.17	6.02	9.25	2.78	63.56
Ⓕ VGG16-SelfAtt	16.55	30.03	24.39	25.90	0.01	27.37	14.01	2.96	7.07	0.84	85.71
Ⓖ VGG16-GTIngr	26.58	39.57	42.16	39.13	6.77	34.30	19.25	6.52	9.85	3.37	63.63

Table 4.1: Metrics in percentage for the different model configurations, computed on the validation set

The baseline model achieved surprisingly good results for the ingredients with an F₁ score of 32.18%. Configuration Ⓐ, which does not use the ingredient features for generating the title and instructions, performs significantly better for all metrics than the baseline model. Configuration Ⓑ additionally applies attention to the ingredient features while generating the recipe title and instructions text. It shows that this procedure significantly increases the ingredients/instructions coherence score, i.e. the model is better in mentioning all predicted ingredients in the instructions text. Replacing the VGG-16 encoder with a more complex backbone for the image encoder could not improve the results, as shown by the results of models Ⓒ, Ⓓ and Ⓔ. Therefore, for all subsequent experiments, the VGG-16 model was used. Springenberg et al. (2018) proposed to use self-attention for the extracted image features. However, the results from configuration Ⓕ show that the model was not able to accurately predict the ingredients and recipe title. Configuration Ⓖ, which uses the ground truth ingredients to generate the title and instructions, achieves slightly better performance for the ingredients and instructions task compared to configuration Ⓑ.

To better interpret the ingredient/instructions coherence scores, the metric was also computed using both the ground truth ingredients and instructions text. The result of 58% comes at a surprise because it shows that all models except the baseline model and configuration Ⓐ are better in capturing the ingredient/instructions coherence than human writers. Further inspection revealed that humans sometimes refer to the ingredients as a whole when writing the instructions text, such as *"Alle Zutaten zusammen mischen"*, which leads to this lower score.

4.1.2 Number of Trainable Parameters

The number of trainable parameters per model are given in table 4.2. Configurations Ⓐ and Ⓑ have the same number of parameters as they only differ in the training procedure. Replacing the image encoder with more complex networks increases the number of parameters to a maximum of 80.6M for configuration Ⓓ. Adding two self-attention layers to the VGG-16 network increases the number by 6M for configuration Ⓕ. The additional embedding layer for configuration Ⓖ requires 0.7M more parameters compared to configuration Ⓑ.

Model	Parameters
Ⓐ VGG-16-NoIngr	71'025'831
Ⓑ VGG-16	71'025'831
Ⓒ VGG-19	76'335'527
Ⓓ ResNet-50V2	80'616'935
Ⓔ Inception-V3	78'865'927
Ⓕ VGG-16-SelfAtt	77'330'599
Ⓖ VGG-16-GTIngr	71'709'351

Table 4.2: Number of parameters for the different model configurations

4.1.3 Model Selection

The analysis of the metrics revealed that the models with the VGG-16 image encoder achieved the best results. Additionally applying attention to the ingredient features for generating the title and instructions text led to an increased ingredients/instructions coherence score. Configuration ③ achieves good results for all three tasks. However, it requires an additional ingredients encoder in the form of an embedding layer. The model ② achieves comparable results while requiring less parameters. Another interesting property is that a forward pass during training does not require the ground truth ingredients as the instructions are predicted by means of the ingredient features generated from the ingredients decoder. Consequently, model ② was selected for further evaluation.

4.1.4 Shuffling Ingredients

Lee et al. (2020) suggests to shuffle the ingredients during training. To check whether the performance of configuration ② can be increased by using this data augmentation strategy, the model was retrained by shuffling the ingredients. The results in table 4.3 reveal that the precision of the ingredients can be significantly improved, whereas the recall and the F_1 score are diminished. The performance of the title and instructions generation was reduced for all metrics.

Shuffling	Ingredients				Title	Instructions					
	IoU	Pr	Re	F_1	Bl_1	Bl_1	Bl_2	Bl_4	Ro_2	Ci_4	I/I
No	26.15	39.05	41.69	38.62	7.12	32.28	17.97	6.08	9.54	3.28	61.71
Yes	25.15	52.7	30.39	37.13	5.96	23.68	13.35	4.69	9.25	3.08	55.8

Table 4.3: Metrics of the model trained with and without shuffling the ingredients, computed on the validation set in percentage

4.1.5 Beam Search

To check whether the beam size algorithm improves the results, the metrics were calculated for different beam sizes. The results are presented in table 4.4, where the best metrics for each beam sizes are highlighted. The ingredient precision increases with increasing beam size, whereas almost all other metrics decrease. Therefore, the final evaluation on the test set was conducted using greedy search.

Beam Size	Ingredients				Title	Instructions					
	IoU	Pr	Re	F_1	Bl_1	Bl_1	Bl_2	Bl_4	Ro_2	Ci_4	I/I
1 (greedy)	26.15	39.05	41.69	38.62	7.12	32.28	17.97	6.08	9.54	3.28	61.71
2	25.14	39.65	38.41	37.27	6.03	31.61	17.69	6.06	9.59	3.30	62.66
3	24.93	40.32	37.21	36.94	5.36	31.25	17.45	6.03	9.51	3.26	63.37
5	24.35	41.17	35.16	36.16	4.59	29.45	16.32	5.63	9.11	2.95	63.21

Table 4.4: Metrics in percentage for the selected model configuration, computed on the validation set for different beam sizes

4.2 Final Evaluation

This section presents the results of the selected model ② measured on the test set. The model was evaluated using greedy search. The first row in table 4.5 reports the results of the complete pipeline and reflects the metrics computed on the validation set. Further evaluation showed that in 58% of the cases, the ground truth instructions text contains more words than the prediction. However, too few ingredients were predicted in only 36% of the cases.

To check how much the correct prediction of the ingredients influences the instructions decoder, a ceiling analysis was conducted where the model was evaluated using the ground truth ingredients. The results in the second row show that the metrics for the title prediction are doubled. Therefore, the model heavily relies on the correct prediction of the ingredients to generate the title correctly. In contrast, the results of the instructions could not be improved as much.

To further check how much influence the ingredient features have, the model was evaluated without the use of any ingredients, i.e. the instructions decoder was only conditioned on the image features. The results on the third row show that the metrics are significantly decreased when only using the image features. Especially the low ingredients/instructions coherence suggests that the model is not able to mention the correct ingredients in the instructions text.

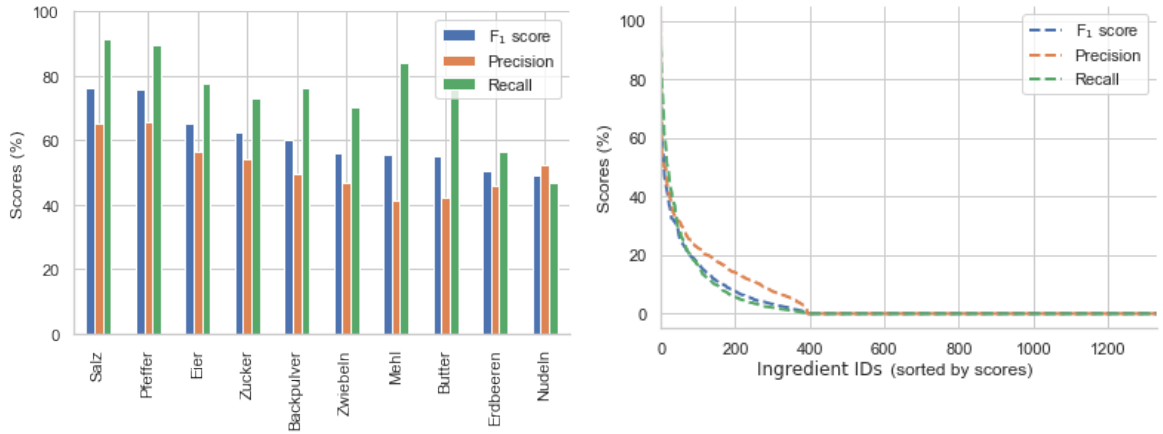
Evaluation	Ingredients				Title	Instructions					
	IoU	Pr	Re	F ₁	Bl ₁	Bl ₁	Bl ₂	Bl ₄	Ro ₂	Ci ₄	I/I
Complete	26.14	39.08	41.68	38.62	7.17	32.4	18.09	6.15	9.54	3.41	61.67
GT Ingr	-	-	-	-	14.73	35.69	21.36	8.46	13.19	8.89	55.53
No Ingr	-	-	-	-	3.22	7.95	4.13	1.19	5.11	0.66	5.76

Table 4.5: Metrics of the selected model configuration, measured on the test set and displayed in percentage

In comparison, Salvador et al. (2018) reported an F₁ score of 48.26% as well as an IoU of 31.8% for the ingredients prediction task. Unfortunately, they did not report the performance of the title and instructions prediction tasks. They trained their models on the Recipe1M dataset that contains 1M recipes.

4.2.1 Ingredients

To identify which ingredients were particularly well predicted, label-based metrics were calculated for the ingredient task. Figure 4.1a shows the metrics for the top 10 ingredients ranked by their F₁ scores. Interestingly, the top 4 ingredients (*Salz*, *Pfeffer*, *Eier*, *Zucker*) correspond to the ranking based on the number of occurrences in the dataset. To get an understanding of how influential the number of occurrences in the training set is for the performance, the Pearson correlation between the occurrences and the F₁ score was calculated. It exhibits a high correlation of 0.71, which also indicates that the model fails to accurately predict ingredients that occur only rarely. Figure 4.1b visualizes the metrics per ingredient, where the ingredients in the *x*-axis are sorted by the scores. It can be noticed that there is an exponential decrease in the metrics. Only 394 achieve an F₁ score above 0, accounting for 29.6% of all ingredients.



(a) Scores of the top 10 ingredients (sorted by F₁ score)

(b) Scores per ingredient (sorted by scores)

Figure 4.1: Label-based metrics for the ingredient prediction task

4.2.2 Qualitative Evaluation

To compute the final metrics, the results were averaged over the different samples either by computing the mean (macro-averaging) or by counting the n -gram matches (micro-averaging). In order to analyse which recipes perform particularly well or poorly, the metrics were additionally computed per sample. Each example was ranked by combining the F_1 score from the ingredients with the $BLEU_1$ score from the instructions in the following manner:

$$\text{score} = \frac{BLEU_1 - BLEU_1^{(\min)}}{BLEU_1^{(\max)} - BLEU_1^{(\min)}} + \frac{F_1 - F_1^{(\min)}}{F_1^{(\max)} - F_1^{(\min)}} \quad (4.1)$$

The scores were scaled by applying min-max normalization since the F_1 scores from the ingredients are on average higher than the $BLEU_1$ scores from the instructions. Finally, the top and worst 100 recipes were obtained by sorting this combined score.

Best Results

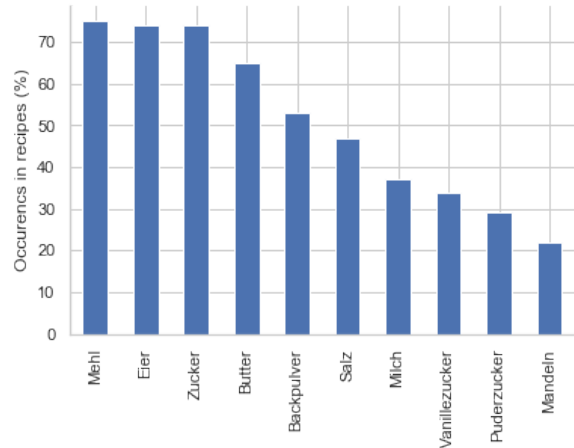
By examining the images shown in figure 4.2a of the 100 recipes with the best results, it can be assumed that most of them are cakes. To get a deeper understanding for which recipes the model performed exceptionally well, a word cloud of these recipes titles was generated (figure 4.2b) as well as an ingredient frequency plot (figure 4.2c). It reveals that there are indeed a lot of cakes since the words "Marmorkuchen" and "Gugelhupf" appear frequently in the recipe titles. Looking at the top 10 ingredients for these recipes further suggests this assumption, as they are typical ingredients for cakes.



(a) Images of the top 100 recipes




(b) Word cloud of the ground truth titles




(c) Top 10 ingredients

Figure 4.2: 100 recipes that were predicted the most accurately

Figure 4.3 shows two examples from the test set that performed exceptionally well. Both ground truth and prediction are visible for each example. Ingredients are written in bold if they occurs in the ground truth and prediction simultaneously. The first example shows an apple pie for which the model even predicted the correct baking time (50 minutes). However, the model generated the wrong temperature. In the second example, the model predicted the ingredient *Zucchini*, which apparently does not appear in the ground truth recipe. However, a person would argue that this ingredient is actually present in the image.

	
Ground Truth	Prediction
Apfelmuchen mit Puderzucker	Apfelmuchen mit Mandeln
Puderzucker Butter Mehl Mandeln Zucker Backpulver Apfel Brösel Milch Zitronensaft Eier	Puderzucker Butter Mehl Mandeln Zucker Backpulver Apfel Milch Zitronensaft Eier
Den Backofen auf 170°C Ober- und Unterhitze vorheizen. Eine Springform (ca. 24 cm) mit Butter auspinseln und mit Bröseln ausstreuen. Für den Teig die Butter mit dem Zucker und Salz schaumig rühren. Unter weiterem Rühren nach und nach die Eier zugeben. Das mit Backpulver und den Mandeln vermischte Mehl und die Milch abwechselnd unterrühren, so dass ein geschmeidiger Teig entsteht. In die Form füllen und glatt streichen. Die Äpfel schälen, vierteln und das Kernhaus herausschneiden. Die Viertel auf der gewölbten Seite jeweils mehrmals einschneiden. Mit dem Zitronensaft beträufeln und auf dem Teig auslegen. Leicht eindrücken und im Ofen ca. 50 Minuten goldbraun backen (Stäbchenprobe). Sollte der Kuchen zu dunkel werden, mit Alufolie abdecken. Den Kuchen aus dem Ofen nehmen und auskühlen lassen. Mit Puderzucker bestäuben.	Den Ofen auf 180°C Unter- und Oberhitze vorheizen. Eine Springform mit Butter auspinseln und mit Mehl ausstreuen. Die Äpfel schälen, vierteln, das Kernhaus herausschneiden und die Viertel in schmale Spalten schneiden. Mit dem Zitronensaft beträufeln. Die Butter mit dem Zucker cremig rühren. Nach und nach die Eier zugeben und unterrühren. Das Mehl mit den Mandeln und dem Backpulver darüber streuen und alles mit der Milch zu einem glatten Teig verrühren. In die Form füllen und glatt streichen. Die Apfelspalten fächerartig auf den Teig legen und leicht in den Teig drücken. Im Ofen ca. 50 Minuten backen (Stäbchenprobe). Den Kuchen aus dem Ofen nehmen, abkühlen lassen, aus der Form lösen und auf einem Kuchengitter vollständig auskühlen lassen. Mit Puderzucker bestäuben.

(a) Example 1 that performed exceptionally well

	
Ground Truth	Prediction
Griechischer Hähnchenaufbau	Hähnchenschenkel mit mediterranem Gemüse
Oliveöl Kartoffeln Hähnchenschenkel Salz Paprika-schote Paprika-pulver Knoblauch Zwiebeln	Thymian Oliveöl Kartoffeln Hähnchenschenkel Salz Paprika-schote Zucchini Paprika-pulver Knoblauch Zwiebeln Pfeffer
Die Kartoffeln waschen, schälen und in dicke Scheiben schneiden. Die Zwiebeln schälen und ebenfalls in Scheiben schneiden. Die Paprika putzen waschen und in Streifen schneiden. Das Gemüse in eine Auflaufform oder aufs Backblech geben, mit etwas Oliveöl beträufeln mit Salz, Pfeffer, Paprika würzen und mit den Händen vermischen. Dazu evt. ganzen oder klein geschnittenen Knoblauch. Die Hähnchenkeulen waschen, trocken tupfen und mit Salz, Pfeffer, Paprika würzen und auf das Gemüse legen. Dann für ca. 90-100 Minuten bei 200 Grad in den vorgeheizten Backofen. Zwischendurch einmal wenden. Das kann man noch endlos variieren mit anderem Gemüse, wie Zucchini, Pilze, Oliven und so weiter.	Die Hähnchenschenkel waschen, trocken tupfen und mit Salz, Pfeffer und Paprikapulver würzen. Die Hähnchenschenkel in eine Auflaufform legen. Die Kartoffeln schälen, waschen und in Spalten schneiden. Die Zucchini waschen, putzen und in Scheiben schneiden. Die Paprika waschen, putzen und in mundgerechte Stücke schneiden. Die Zwiebel abziehen und in Spalten schneiden. Den Knoblauch abziehen und fein hacken. Das Gemüse in einer Schüssel mit dem Oliveöl und dem Rosmarin vermischen. Mit Salz und Pfeffer würzen. Die Gemüsemischung auf den Hähnchenschenkel verteilen. Die Auflaufform in den vorgeheizten Backofen schieben und bei 180°C etwa 45 Minuten garen. Nach etwa 30 Minuten die Hähnchenschenkel mit dem Gemüse servieren. Dazu schmeckt ein frisches Baguette.

(b) Example 2 that performed exceptionally well

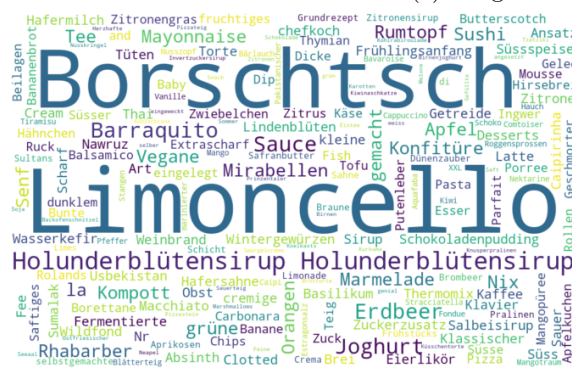
Figure 4.3: Examples from the test set that performed exceptionally well

Worst Results

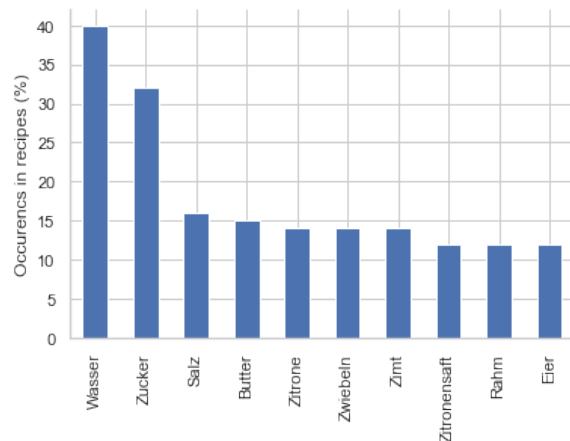
Examining the worst results seems not that straightforward, as the images of the worst 100 results depicted in figure 4.4a contain a wide variety of dishes. The word cloud of the ground truth titles (4.4b) shows that recipes for *Borschtsch* and *Limoncello* are hard to predict. Other than that, it is hard to identify any patterns because there are many different recipe names in the word cloud. The bar plot of the ingredients depicted in figure 4.4c does not help to identify certain types of dishes that the model struggles the most with.



(a) Images of the top 100 recipes



(b) Word cloud of the ground truth titles



(c) Top 10 ingredients

Figure 4.4: 100 recipes that were most inaccurately predicted

On average, these ground truth instructions contain around 324 words. In contrast, the predictions contain only 22 words in average, which means that the model generates too short recipes. Manual inspection showed that a lot of these recipes contain the sentence *"Alle Zutaten miteinander verrühren"*. Therefore, it can be assumed that when the model is not able to come up with a certain recipe, it simply generates a text saying that all ingredients should be mixed.



Figure 4.5: A word cloud for the prediction of the instruction text of the worst results


CHAPTER 4. RESULTS

The word cloud in figure 4.5 supports this assumption, as a lot of collocations occur that say mixing up ingredients. Apart from that, it is noticeable that there are several collocations that are used in baking instructions. From this it can be concluded that the model overfitted on baking recipes.

In figure 4.6 two examples are visible that performed exceptionally poor. Both predicted instructions start with mixing all ingredients together.

Ground Truth			Prediction	
Senf (Grundrezept)		Zitronendressing		
Speiseöl Wasser Zucker	Die Senfkörner mit dem Salz und dem Zucker verrühren, die Flüssigkeiten zufügen, ebenso ein paar Tropfen Speiseöl. Mit dem elektrischem Handrührgerät so lange verrühren, bis der Senf die richtige Konsistenz hat. In Schraubgläser füllen und im Kühlschrank aufbewahren. Der Senf ist Anfangs sehr scharf, er wird aber innerhalb der nächsten Tage etwas milder.	Zitronensaft Oliveneöl	Alle Zutaten miteinander vermischen und pikant abschmecken.	

(a) Example 1 that performed exceptionally poor

		
<h2>Ground Truth</h2>		<h2>Prediction</h2>
<h3>Pasta - Teig</h3>		<h3>Glühwein</h3>
<p>Salzwasser Oliveneöl Wasser Salz Weizenmehl Eier</p>	<p>Auf eine Arbeitsfläche (z. B. grosses Arbeitsbrett) das Mehl geben. In das Mehl eine Vertiefung drücken und die geschlagenen Eier, das Salz und das Olivenöl giessen. Mit einer Gabel diese Mischung aus Eiern und Olivenöl verrühren und dabei immer mehr Mehl vom Rand mit hineinrühren. Sobald die Mischung dicker wird, mit den Händen das Mehl weiter untermischen. Wenn der Teig zu trocken ist, ein wenig Wasser hinzuarbeiten, aber nicht zu viel. Der Teig muss eine klumpige und grobe Masse sein und darf nicht zu feucht sein. Den Teig etwa eine Viertelstunde mit den Händen kräftig weiter wie Brotteig kneten. Der Teig sollte am besten von sich weg mit den Handballen geknetet werden, anschliessend wieder zu sich herziehen und dann wieder mit den Handballen wie Brotteig von sich "wegkneten". Nach einer Viertelstunde durchkneten sollte der Teig glatt und geschmeidig sein. Aus dem Teig zwei Kugeln formen und noch etwa eine Viertelstunde ruhen lassen. Anschliessend kann der Teig bearbeitet werden, entweder mit einer Pastamaschine, oder, wer keine hat, der rollt mit einem Nudelholz jede Kugel für sich auf einem grossen Brett zu einem sehr dünnen Teig aus. Nun kann dieser Teig entweder für Ravioli weiterverwendet werden, oder, so mache ich es gerne, ich mache anschliessend Tagliatelle daraus. Dafür schneide ich diesen dünnen Pastafaden einmal durch, rolle ihn anschliessend auf und schneide dünne Streifen, ca. 1/2 cm, davon ab. Diese werden anschliessend auseinander gewickelt und auf ein sauberes bemehltes Geschirrtuch gegeben. Die Pasta in kochendem Salzwasser al dente kochen (ca. 2 - 3 Minuten). Dieses Rezept ist für zwei Personen (gute Esser) als Hauptmahlzeit berechnet.</p>	<p>Zimt Gewürznelken Muskat Ingwer Tee Zimtstange Zucke Piment Anis</p> <p>Alle Zutaten in einen Topf geben und erhitzen, nicht kochen. In saubere Gläser füllen und sofort verschliessen. Der Glühwein ist sehr heiss und kann sofort getrunken werden.</p>

(b) Example 2 that performed exceptionally poor

Figure 4.6: Examples from the test set that performed exceptionally poor

Chapter 5

Discussion

Chapter 5 discusses the results of this project and provides an outlook regarding future work. Moreover, it shows whether the evaluated models are applicable in the medical domain.

5.1 Outlook

There are several aspects that could be investigated in order to improve the current work. In addition, some further studies could be carried out on the basis of the results of this project.

Improve Data Cleaning The analysis of the data showed that some recipes contain images from earlier stages of the preparation process. This kind of images are not helpful for training the model, because they do not contain the full information about the result. For example, baking recipes could have pictures showing the dough in its unbaked form. Removing such images could improve the performance of the model. Another issue is the number of unique ingredients. Even though it could be reduced from 1M to roughly 1.3k by applying a rather complex cleaning pipeline, manual inspection revealed that there are still several duplicates. Evaluation of the model demonstrated that only 394 ingredients achieve an F_1 score above 0. Investing some additional time in the cleaning process could therefore further improve the performance of the model.

Different Attention Mechanisms To allow the instructions decoder to pay attention to both the image features and ingredients at the same time, the corresponding features were concatenated into a single feature vector. Salvador et al. (2018) tried two other strategies: applying attention independently and then concatenating the feature vectors, as well as applying attention sequentially. It would be worth investigating how well these approaches work with the collected recipe dataset.

Predict Recipe Title and Instructions Separately The proposed recipe model uses the multi-field learning technique to predict the title and instructions text at the same time. This way, the number of parameters required can be significantly reduced, as only one decoder network is employed. However, concatenating the two outputs increases the length of the corresponding target values, making it more difficult for the model to maintain long-term dependencies. The recipe title could be predicted separately by applying attention to the image, ingredient and instruction features. Since both ingredient and instruction are then required to predict the title, this could act as a semantic regularization that helps the model to better learn the joint representation, leading to increased performance. Such a regularization loss was used by Salvador et al. (2017) to train a recipe retrieval model.

Self-Critical Sequence Training To improve the performance of the model, the self-critical sequence training technique from Rennie et al. (2016) could be applied after training the model. This way the exposure bias could be diminished while improving metrics like BLEU or the CIDEr score. However, because the model has multiple outputs, further research needs to be carried out on how this training technique could be applied in this case.

5.2 Adaption to Medical Domain

Since this project is motivated by a problem in the medical domain, this section shortly describes whether the proposed methods for recipe generation can be adapted for generating medical reports.

The topic of medical image captioning has been explored by several researchers. Recently, Jing et al. (2018) introduced a model for generating medical reports based on radiology images. They used the IU Chest X-Ray dataset (Demner-Fushman et al., 2016) that consists of 7'470 images paired with medical reports. Such a report contains tags, impressions and findings. Tags are a list of keywords, whereas impressions can be viewed as a conclusion or topic sentence of the report. A written description of the diagnosis is included in the findings. The output of their medical image captioning system is a set of tags as well as a long diagnosis text. Therefore, they concatenate the impression and findings text. They treat the prediction of tags as a multi-label classification task and the prediction of the diagnosis text as a text generation task. First, a pre-trained VGG-19 network is used to extract visual features from the image, which are then used by a multi-label classification network to predict the tags. Both the image features and the tags are then used to generate the diagnosis text. Their proposed architecture is based on the hierarchical model introduced by Krause et al. (2017) (see section 2.7.5), that uses a sentence LSTM and a word LSTM to produce the output text. Figure 5.1 illustrates their proposed model.

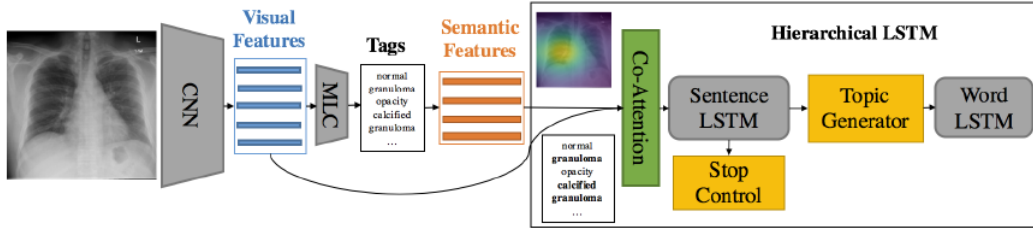

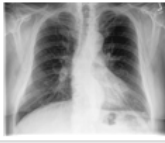


Figure 5.1: An overview of the chest x-ray medical report generation model proposed by Jing et al. (2018)

Dataset and architecture are very similar to the proposed recipe generation model. On the one hand, the tags can be seen as a proxy for the ingredients, which in both cases are predicted using a multi-label classification model. On the other hand, the diagnosis text corresponds to the recipe instructions that are generated using a text generation model. The impression field describes the topic of a report, thus it can be predicted in the same manner as the recipe title. Figure 5.2 compares two examples from both datasets. The main difference of the model architectures is that the recipe model uses transformer networks, whereas the medical report generation model from Jing et al. (2018) uses LSTMs.

Input Image 		Input Image 	
Title: Spaghetti mit Tomatensauce		Impression: No acute cardiopulmonary process.	
Ingredients: Spaghetti Tomaten Olivenöl Wasser Salz	Instructions: Erst Zwiebeln, Tomaten und Knoblauch würfeln. Zwiebeln in heißem Olivenöl anbraten, dann mit etwas Rotwein ablöschen und bei niedriger Hitze köcheln lassen. Dann Tomaten und Knoblauch zu den Zwiebeln geben. Mit etwas Salz und Pfeffer abschmecken. Nudeln in Salzwasser "al dente" kochen.	Tags: Calcified granuloma Thoracic vertebrae Aorta thoracic	Findings: Heart size is normal. There is tortuosity of the thoracic aorta, stable compared with prior. No focal airspace disease or effusion. No pleural effusions or pneumothoraces. Degenerative changes in the thoracic spine.

(a) An example from the recipe dataset

(b) An example from the IU chest x-ray dataset

Figure 5.2: Comparison of the recipe and IU chest x-ray data

Liu et al. (2019) explain that medical reports often have a template-based structure and the accurate creation of the clinical description is the top priority. However, in generic natural language generation, readability is preferred over accuracy. Cooking recipes share both properties, as an accurate description of the instruction text is crucial for a cook to successfully prepare a dish. Analysis of the instructions text showed that recipes have an inherent template-based structure as well. For example, when a recipe involves baking, the sentence *"Backofen auf x Grad vorheizen"* appears almost every time. In general, generating medical reports might be even easier than cooking recipes because the vocabulary is more standardized. Jing et al. (2018) describe that the vocabulary of the IU Chest X-Ray dataset consists of 1'915 unique words. In contrast, the vocabulary of the crawled recipe dataset consists of 531'809 unique words. Moreover, to the author's best knowledge, there are not that many synonyms in the medical field. In cooking, however, there are numerous synonyms for ingredients such as *Rüebli* and *Karotten*.

5.3 Conclusion

Motivated by the generation of medical reports, an inverse cooking system was implemented, trained and evaluated. A corpus of 0.9M German recipes and 1.3M images was acquired through crawling five cooking websites. Since most of the recipes origin from community cooking platforms, an intensive data cleaning pipeline was developed. Applying this pipeline reduced the number of unique ingredients from 1M to 1.3k. The implemented multi-task model generates a list of ingredients, a recipe title and a cooking instructions text based on an image of a dish. Experiments demonstrated that the quality of the instructions text can be significantly improved if the model can simultaneously pay attention to the image and the ingredient features while generating the output. Comparison of the proposed inverse cooking system to medical image captioning systems from the literature showed that the architectures and datasets share similarities. A medical report often consists of tags, which are a proxy for the ingredients. Moreover, some reports contain an impression section, which summarizes the diagnosis. This can be seen as a proxy for the recipe title. Lastly, the findings section of a medical report is a long text describing the diagnosis and serves as a proxy for the cooking instructions. Altogether, the challenges in the recipe generation align well with those in medical image captioning. For both domains, generating the outputs involves making decisions based on assumptions about incomplete data and adapting to rare situations using knowledge from similar past experiences. Given these similarities, it is expected that the proposed model architecture can be adapted for generating medical reports in the future.

Appendix A

Appendix

A.1 Demonstration App

A demonstration app that allows to try out the inverse cooking model was implemented. It is available at **image-to-recipe.abiz.ch**. Based on an uploaded picture of a dish, the app generates a list of ingredients, a title and a recipe instruction text. The beam size can be selected prior uploading the picture. When hovering over the predicted ingredients, the parts of the image are highlighted to which the model paid attention to. Also, the self-attention scores are highlighted. Moreover, hovering over the instructions visualizes the attention to the predicted ingredients. The example depicted in figure A.1 demonstrates the output where a picture of a carrot cake was uploaded. Interestingly, it shows that the model paid attention to the word *"Karotten"* when generating the word *"Möhren"*, which is actually a synonym. To speed up inference, caching in the self-attention of the ingredients and instructions decoder was implemented. At each time step t , the previously computed key **K** and value **V** matrices are stored.



Figure A.1: An example output of the demonstration app

A.2 Code

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

A.2.1 Jupyter Notebooks

The Jupyter notebooks listed below were implemented to either clean or preprocess the data, or to create the plots that were displayed throughout this report.

00_Data_Wrangling_Subsets.ipynb To concat all data from the scraped cooking websites into one dataset.

01_Data_Cleaning.ipynb Performs the data cleaning described in section 3.1.1.

02_Data_Analysis.ipynb Performs data analysis for the cleaned data described in section 3.1.2.

03_Data_Preprocessing.ipynb To tokenize and preprocess the data so it can be used for training the model.

04_Data_Pipeline_Test.ipynb Notebook to test the data pipeline.

05_Train_Embeddings.ipynb Retrain FastText embeddings that were used for data cleaning and model evaluation.

06_Evaluation.ipynb To generate the plots and metrics described in section 4.2.

A.2.2 Recipe Model

To train the model, the script `train.py` was implemented that starts the training process for the recipe model.

```
$ python3 train.py
--data_dir="/app/data"
--work_dir="/app/work"
--max_tokens_per_batch=8000
--run_name="recipe_model"
```

After training the recipe model, the script `eval.py` can be executed to calculate the metrics by specifying the correct checkpoint directory.

```
$ python3 eval.py
--data_dir="/app/data"
--dataset="val"
--max_tokens_per_batch=8000
--checkpoint_dir="/app/work/checkpoint/recipe_model"
```

List of Figures

2.1	A graphical illustration of an MP neuron proposed by McCulloch and Pitts (1943) . . .	4
2.2	A graphical illustration of a perceptron proposed by Rosenblatt (1958)	4
2.3	An example of an MLP with one hidden layer and two inputs	5
2.4	The sigmoid activation function	5
2.5	The tanh activation function	6
2.6	The ReLU activation function	6
2.7	The leaky ReLU activation function with $\alpha = 0.1$	7
2.8	The ELU activation function with $\alpha = 1$	8
2.9	The GELU activation function	8
2.10	A visualization of a residual block by He et al. (2015)	12
2.11	A CNN layer with size 3×3 , stride = 2 and padding = 1 applied to an input of dimension 5×5	14
2.12	Illustration of the dense connectivity of feed-forward networks	14
2.13	Illustration of the sparse connectivity of convolutional networks	14
2.14	An example of max-pooling and average-pooling with a pooling size 2×2 and stride 2 .	15
2.15	An inception module introduced with the GoogLeNet by Szegedy et al. (2014)	16
2.16	Visualization of the ResNet (He et al., 2015) and the ResNeXt (Xie et al., 2016) architecture	16
2.17	Visualization of an RNN unfolded over time	17
2.18	Visualization of a simple RNN cell by Colah (2015)	17
2.19	Visualization of a GRU cell by Colah (2015)	18
2.20	Visualization of a LSTM cell by Colah (2015)	19
2.21	Visualization of a bidirectional RNN unfolded over time	19
2.22	The proposed Word2Vec models, visualized by Rong (2014)	25
2.23	An illustration of the extraction of the flair embedding for the word <i>Washington</i> from the original flair paper by Akbik et al. (2018)	26
2.24	An illustration of the image-to-text retrieval method proposed by Farhadi et al. (2010) .	28
2.25	The image-to-text model proposed by Vinyals et al. (2014)	29
2.26	The image-to-text model with attention proposed by Xu et al. (2015)	30
2.27	The image-to-text model with semantic attention	32
2.28	The architecture of the proposed dense captioning model by Johnson et al. (2016) . . .	33
2.29	An overview of the proposed model by Krause et al. (2017)	33
2.30	The architecture of the transformer based image-to-text model proposed by Zhu et al. (2018)	35
2.31	A visualization of the computation graph of the multi-head attention in Vaswani et al. (2017)	36
2.32	Multi-level supervision for three transformer decoder layers proposed by Zhu et al. (2018)	37
2.33	An example of a beam search decoder with beam size 2	38
2.34	An example where a sentence is transformed into a semantic scene graph, illustrated in Anderson et al. (2016)	43
2.35	An illustrations of the WMD metric	44
2.36	Comparison between teacher forcing during training and free running during inference time	45
2.37	An illustration of the professor forcing technique proposed by Goyal et al. (2016)	46
2.38	Examples from the Food-101 dataset by Bossard et al. (2014)	47
2.39	The joint neural embedding model for the task of image-to-recipe retrieval proposed by Salvador et al. (2017)	48
2.40	The image-to-recipe retrieval approach used by Chen et al. (2017a)	48
2.41	An illustration of the inverse cooking model proposed by Salvador et al. (2018)	49

LIST OF FIGURES

3.1	The histogram of the instruction lengths in number of characters	52
3.2	Maximum instructions length vs. recipes to drop	52
3.3	Ingredient frequencies before cleaning	53
3.4	Elbow plot of the minimum ingredient occurrence	54
3.5	Boxplot of the number of ingredients	55
3.6	Ingredient frequencies after cleaning	56
3.7	Cumulative sums of the ingredient occurrences in percentage	57
3.8	A word cloud of the instruction texts that contains the most frequent words and collocations	57
3.9	Number of images per recipe	58
3.10	Images of the recipe <i>Lasagne</i> from the dataset	59
3.11	Examples of data augmentation	61
3.12	Tokenization of the recipe title and instructions text	62
3.13	An illustration of the baseline model	65
3.14	An overview of the proposed recipe generation model	66
3.15	Global max-pooling applied to the generated ingredients	67
3.16	The implemented β_2 and learning rate scheduler	69
4.1	Label-based metrics for the ingredient prediction task	74
4.2	100 recipes that were predicted the most accurately	75
4.3	Examples from the test set that performed exceptionally well	76
4.4	100 recipes that were most inaccurately predicted	77
4.5	A word cloud for the prediction of the instruction text of the worst results	77
4.6	Examples from the test set that performed exceptionally poor	78
5.1	An overview of the chest x-ray medical report generation model proposed by Jing et al. (2018)	80
5.2	Comparison of the recipe and IU chest x-ray data	80
A.1	An example output of the demonstration app	82

List of Tables

2.1	BLEU score example for unigram precision	40
3.1	Different recipe sources with number of recipes	51
3.2	Recipe titles with its number of occurrences	56
3.3	Statistics of the recipe titles	56
3.4	Frequent itemsets of ingredients	57
3.5	Recipes with the highest number of images	58
3.6	Number of samples in training, validation and test set	60
3.7	Setup of the image encoder	67
3.8	Setup of ingredients decoder	67
3.9	Setup of ingredients decoder loss	68
3.10	Setup of the instructions decoder	69
3.11	Optimizer configuration of the recipe model	70
4.1	Metrics in percentage for the different model configurations, computed on the validation set	72
4.2	Number of parameters for the different model configurations	72
4.3	Metrics of the model trained with and without shuffling the ingredients, computed on the validation set in percentage	73
4.4	Metrics in percentage for the selected model configuration, computed on the validation set for different beam sizes	73
4.5	Metrics of the selected model configuration, measured on the test set and displayed in percentage	74

List of Algorithms

1	Gradient descent algorithm	9
2	Momentum algorithm	10
3	RMSProp algorithm	10
4	Adam algorithm	10
5	Batch normalization	11
6	Layer normalization	11
7	Compute the next hidden state \mathbf{h}_t for a GRU cell	18
8	Compute the next hidden state \mathbf{h}_t for a LSTM cell	19
9	Compute the output of the CNN decoder using GLU activations	34

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.
- Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules. In *Proc. of 20th Intl. Conf. on VLDB*, pages 487–499.
- Akbik, A., Blythe, D., and Vollgraf, R. (2018). Contextual string embeddings for sequence labeling. In *COLING*.
- Anderson, P., Fernando, B., Johnson, M., and Gould, S. (2016). Spice: Semantic propositional image caption evaluation. In *ECCV*.
- Aneja, J., Deshpande, A., and Schwing, A. (2018). Convolutional image captioning. In *Proceedings - 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2018*, Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pages 5561–5570. IEEE Computer Society. 31st Meeting of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2018 ; Conference date: 18-06-2018 Through 22-06-2018.
- Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization.
- Baddam, S. R., Gollapudi, S. S., and Edupuganti, S. K. (2020). Neuralcook – image2ingredients and cooking recommendation using deep learning.
- 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.
- Bengio, S., Vinyals, O., Jaitly, N., and Shazeer, N. (2015). Scheduled sampling for sequence prediction with recurrent neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1, NIPS’15*, page 1171–1179, Cambridge, MA, USA. MIT Press.
- 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.
- Bossard, L., Guillaumin, M., and Van Gool, L. (2014). Food-101 – mining discriminative components with random forests. In Fleet, D., Pajdla, T., Schiele, B., and Tuytelaars, T., editors, *Computer Vision – ECCV 2014*, pages 446–461, Cham. Springer International Publishing.
- Bravin, M. (2020). Notam smartification.
- Chen, J.-j., Ngo, C.-W., and Chua, T.-S. (2017a). Cross-modal recipe retrieval with rich food attributes. In *Proceedings of the 25th ACM International Conference on Multimedia, MM ’17*, page 1771–1779, New York, NY, USA. Association for Computing Machinery.
- Chen, S.-F., Chen, Y.-C., Yeh, J., and Wang, Y.-C. F. (2017b). Order-free rnn with visual attention for multi-label classification.

BIBLIOGRAPHY

- Chen, X., Fang, H., Lin, T.-Y., Vedantam, R., Gupta, S., Dollár, P., and Zitnick, C. L. (2015). Microsoft coco captions: Data collection and evaluation server. *CoRR*, abs/1504.00325.
- 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.
- Church, K. W. and Hanks, P. (1990). Word association norms, mutual information, and lexicography. *Computational Linguistics*, 16(1):22–29.
- 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.
- Deaton, J., Jacobs, A., and Kenealy, K. (2018). Transformer and pointer-generator networks for abstractive summarization.
- Demner-Fushman, D., Kohli, M. D., Rosenman, M. B., Shooshan, S. E., Rodriguez, L., Antani, S., Thoma, G. R., and McDonald, C. J. (2016). Preparing a collection of radiology examinations for distribution and retrieval. *Journal of the American Medical Informatics Association : JAMIA*, 23(2):304–310.
- Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding.
- Elman, J. L. (1990). Finding structure in time. *COGNITIVE SCIENCE*, 14(2):179–211.
- Farhadi, A., Hejrati, M., Sadeghi, A., Young, P., Rashtchian, C., Hockenmaier, J., and Forsyth, D. (2010). Every picture tells a story: Generating sentences from images. volume 6314, pages 15–29.
- Ganesan, K. (2018). Rouge 2.0: Updated and improved measures for evaluation of summarization tasks.
- 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.
- Godbole, S. and Sarawagi, S. (2004). Discriminative methods for multi-labeled classification. In Dai, H., Srikant, R., and Zhang, C., editors, *Advances in Knowledge Discovery and Data Mining*, pages 22–30, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Goldberg, Y. (2016). A primer on neural network models for natural language processing. *J. Artif. Int. Res.*, 57(1):345–420.
- Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. The MIT Press.
- Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc.
- Goyal, A., Lamb, A., Zhang, Y., Zhang, S., Courville, A., and Bengio, Y. (2016). Professor forcing: A new algorithm for training recurrent networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS’16, page 4608–4616, Red Hook, NY, USA. Curran Associates Inc.
- He, C. and Hu, H. (2019). Image captioning with visual-semantic double attention. *ACM Trans. Multimedia Comput. Commun. Appl.*, 15(1).
- 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.

BIBLIOGRAPHY

- 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.
- Jing, B., Xie, P., and Xing, E. (2018). On the automatic generation of medical imaging reports. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2577–2586, Melbourne, Australia. Association for Computational Linguistics.
- Johnson, J., Karpathy, A., and Fei-Fei, L. (2016). Denscap: Fully convolutional localization networks for dense captioning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.
- 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.
- Khomenko, V., Shyshkov, O., Radyvonenko, O., and Bokhan, K. (2017). Accelerating recurrent neural network training using sequence bucketing and multi-gpu data parallelization.
- Kiddon, C., Zettlemoyer, L., and Choi, Y. (2016). Globally coherent text generation with neural checklist models. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 329–339, Austin, Texas. Association for Computational Linguistics.
- Kilickaya, M., Erdem, A., Ikizler-Cinbis, N., and Erdem, E. (2017). Re-evaluating automatic metrics for image captioning. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 199–209, Valencia, Spain. Association for Computational Linguistics.
- Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization.
- Krause, J., Johnson, J., Krishna, R., and Fei-Fei, L. (2017). A hierarchical approach for generating descriptive image paragraphs. In *Computer Vision and Pattern Recognition (CVPR)*.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012a). 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.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012b). 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.
- Kulkarni, G., Premraj, V., Dhar, S., Li, S., Choi, Y., Berg, A. C., and Berg, T. L. (2011). Baby talk: Understanding and generating simple image descriptions. In *CVPR 2011*, pages 1601–1608.
- Kusner, M., Sun, Y., Kolkin, N., and Weinberger, K. (2015). From word embeddings to document distances. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 957–966, Lille, France. PMLR.
- 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.
- LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324.
- Lee, H., Shu, K., Achananuparp, P., Prasetyo, P. K., Liu, Y., Lim, E.-P., and Varshney, L. (2020). Recipept: Generative pre-training based cooking recipe generation and evaluation system.
- 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.

BIBLIOGRAPHY

- Liu, G., Hsu, T. M., McDermott, M., Boag, W., Weng, W.-H., Szolovits, P., and Ghassemi, M. (2019). Clinically accurate chest x-ray report generation.
- Lu, J., Xiong, C., Parikh, D., and Socher, R. (2016). Knowing when to look: Adaptive attention via a visual sentinel for image captioning. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3242–3250.
- Luong, M.-T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation.
- Maas, A. L. (2013). Rectifier nonlinearities improve neural network acoustic models.
- Majumder, B. P., Li, S., Ni, J., and McAuley, J. (2019). Generating personalized recipes from historical user preferences.
- 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.
- Mihaylova, T. and Martins, A. F. T. (2019). Scheduled sampling for transformers. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: Student Research Workshop*, pages 351–356, Florence, Italy. Association for Computational Linguistics.
- Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space.
- Nam, J., Kim, J., Loza Mencía, E., Gurevych, I., and Fürnkranz, J. (2014). Large-scale multi-label text classification — revisiting neural networks. In Calders, T., Esposito, F., Hüllermeier, E., and Meo, R., editors, *Machine Learning and Knowledge Discovery in Databases*, pages 437–452, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Ng, A. (2017). Improving deep neural networks: Hyperparameter tuning, regularization and optimization.
- 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 *EMNLP*.
- Pineda, L., Salvador, A., Drozdal, M., and Romero, A. (2019). Elucidating image-to-set prediction: An analysis of models, losses and datasets.
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners.
- Rahman, M. A. and Wang, Y. (2016). Optimizing intersection-over-union in deep neural networks for image segmentation. In *ISVC*.
- Read, J. (2010). *Scalable Multi-label Classification*. PhD thesis, Hamilton, New Zealand. Doctoral.
- Read, J., Pfahringer, B., Holmes, G., and Frank, E. (2011). Classifier chains for multi-label classification. *Mach. Learn.*, 85(3):333–359.
- Ren, S., He, K., Girshick, R., and Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 91–99. Curran Associates, Inc.
- Rennie, S. J., Marcheret, E., Mroueh, Y., Ross, J., and Goel, V. (2016). Self-critical sequence training for image captioning. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1179–1195.
- Rong, X. (2014). word2vec parameter learning explained.

BIBLIOGRAPHY

- Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386.
- Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2014). Imagenet large scale visual recognition challenge. cite arxiv:1409.0575Comment: 43 pages, 16 figures. v3 includes additional comparisons with PASCAL VOC (per-category comparisons in Table 3, distribution of localization difficulty in Fig 16), a list of queries used for obtaining object detection images (Appendix C), and some additional references.
- Salimans, T. and Kingma, D. P. (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks.
- Salvador, A., Drozdal, M., Giro-i Nieto, X., and Romero, A. (2018). Inverse cooking: Recipe generation from food images.
- Salvador, A., Hynes, N., Aytar, Y., Marín, J., Ofli, F., Weber, I., and Torralba, A. (2017). Learning cross-modal embeddings for cooking recipes and food images. pages 3068–3076.
- Schmidt, F. (2019). Generalization in generation: A closer look at exposure bias. In *Proceedings of the 3rd Workshop on Neural Generation and Translation*, pages 157–167, Hong Kong. Association for Computational Linguistics.
- Schuster, M. and Paliwal, K. (1997). Bidirectional recurrent neural networks. *Trans. Sig. Proc.*, 45(11):2673–2681.
- Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition.
- Sorower, M. S. (2010). A literature survey on algorithms for multi-label learning.
- Springenberg, S., Lakomkin, E., Weber, C., and Wermter, S. (2018). Image-to-text transduction with spatial self-attention.
- Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, ICML’13*, page III–1139–III–1147. JMLR.org.
- Sutskever, I., Vinyals, O., and Le V, Q. (2014). Sequence to sequence learning with neural networks.
- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*.
- Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2015). Rethinking the inception architecture for computer vision.
- Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*.
- Tieleman, H. (2012). rmsprop: Divide the gradient by a running average of its recent magnitude.
- Torrey, L. and Shavlik, J. (2009). Transfer learning. *Handbook of Research on Machine Learning Applications*.
- Tsoumakas, G. and Vlahavas, I. (2007). Random k-labelsets: An ensemble method for multilabel classification. volume 4701, pages 406–417.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need.
- Vedantam, R., Zitnick, C. L., and Parikh, D. (2015). Cider: Consensus-based image description evaluation. In *CVPR*, pages 4566–4575. IEEE Computer Society.

BIBLIOGRAPHY

- Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2014). Show and tell: A neural image caption generator. *arXiv:1411.4555*.
- Wang, J., Yang, Y., Mao, J., Huang, Z., Huang, C., and Xu, W. (2016). Cnn-rnn: A unified framework for multi-label image classification.
- Wang, Q. and Chan, A. B. (2018). Cnn+cnn: Convolutional decoders for image captioning. *ArXiv*, abs/1805.09019.
- Wei, H., Li, Z., and Zhang, C. (2020). Image captioning based on visual and semantic attention. In Ro, Y. M., Cheng, W.-H., Kim, J., Chu, W.-T., Cui, P., Choi, J.-W., Hu, M.-C., and De Neve, W., editors, *MultiMedia Modeling*, pages 151–162, Cham. Springer International Publishing.
- Wei, Y., Xia, W., Huang, J., ni, B., Dong, J., Zhao, Y., and Yan, S. (2014). Cnn: Single-label to multi-label.
- Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Comput.*, 1(2):270–280.
- Wong, S. C., Gatt, A., Stamatescu, V., and McDonnell, M. D. (2016). Understanding data augmentation for classification: When to warp? In *2016 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, pages 1–6.
- 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.
- Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K. (2016). Aggregated residual transformations for deep neural networks.
- Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., Zemel, R., and Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2048–2057, Lille, France. PMLR.
- Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? In *NIPS*.
- You, Q., Jin, H., Wang, Z., Fang, C., and Luo, J. (2016). Image captioning with semantic attention. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4651–4659.
- Yu, J., Li, J., Yu, Z., and Huang, Q. (2019). Multimodal transformer with multi-view visual representation for image captioning.
- Zhu, X., Liu, J., Haipeng, P., and Niu, X. (2018). Captioning transformer with stacked attention modules. *Applied Sciences*, 8:739.