University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Bachelor Thesis

Learning of Sentence Encoding by Using Duplicate Questions from Stackoverflow

Pilsen 2020 Jan Pašek

Declaration

I hereby declare that this bachelor's thesis is completely my own work and that I used only the cited sources.

Pilsen, 30th April 2020

Jan Pašek

Acknowledgement

I would like to thank Ing. Miloslav Konopik, Ph.D. for valuable advice, factual comments and helpfulness in consultation and elaboration of the bachelor thesis.

The work was supported from European Regional Development Fund-Project "CERIT Scientific Cloud" (No. CZ.02.1.01/0.0/0.0/16_013/0001802).

Abstract

This bachelor thesis aims to create a neural network for natural language understanding in an expert domain. Our outcome can significantly improve tasks such as information retrieval or code generation. To address that, the work proposes a neural network architecture utilizing a code encoder in parallel with a commonly used text encoder. Furthermore, the architecture uses a not widely known *f1 loss*, significantly improving results. An important outcome of this work is a vector representation of text stored in hidden layers of the network. We demonstrate our approach on Stackoverflow data utilizing duplicate questions to create a novel dataset, usable beyond the scope of this work. Achieved *f1 score* on the Stackoverflow dataset is 74.1%, improving the *f1 score* by 5.1% compared to a baseline model.

Abstrakt

Tato bakalářská práce se zabývá vývojem neuronové sítě pro porozumění textu v odborném jazyce. Výstupy této práce mohou zlepšit výsledky úloh jako je získávání informací či generování zdrojového kódu. Pro vyřešení této úlohy představujeme novou architekturu neuronové sítě založenou na využití enkodéru kódu společně s textovým enkodérem. Architektura dále využívá nepříliš známou *f1 loss*, která významně zlepšuje dosažené výsledky. Důležitým výstupem této práce je vektorová reprezentace vět, která se nalézá ve skrytých vrstavách neuronové sítě. Navržený přístup je demonstrován na využití duplicitních otázek ze stránky Stackoverflow, ze kterých jsme připravili nový dataset použitelný nad rámec této práce. Pomocí navržené architektury bylo na datasetu dosaženo *f1 score* 74.1 %, což představuje zlepšení o 5.1 % v porovnání s výchozí architekturou.

Contents

1 Introduction **9**

2 Neural Networks **10**

2.1 Neural Network Architecture 10

2.1.1 Artificial Neuron 11

2.1.2 Layer 11

2.1.3 Model 12

2.2 Activation Functions 12

2.2.1 Sigmoid 13

2.2.2 Hyperbolic Tangent (TanH) 13

2.2.3 Rectified Linear Unit (ReLU) 14

2.2.4 Softmax 14

2.3 Classification Using a Neural Network 14

2.4 Loss/Cost Functions 15

2.4.1 Mean Squared Error 15

2.4.2 Cross-Entropy 15

2.4.3 F1 Loss 16

2.5 Training a Neural Network Model 16

2.5.1 Optimizer 17

2.6 Metrics 18

2.6.1 Accuracy 18

2.6.2 Precision, Recall and F1 score 18

2.6.3 Confusion Matrix 19

2.7 Other Neural Network Concepts and Techniques 20

2.7.1 Siamese Neural Networks 21

3 Semantic Vector Representation of Text **22**

3.1 Vector Representation of Words 23

3.1.1 Context-Free Word Representation 23

3.1.2 Contextual Word Representation 24

3.1.3 Subword Embedding 24

3.2 Vector Representation of Sentences 25

3.2.1 Combining Word Embeddings 25

3.2.2 Supervised Representation Learning 25

3.2.3 Unsupervised Representation Learning 26

3.3 Semantic Similarity Task 26

3.3.1 Related Datasets 27
3.3.2 Related Work 28

4 Stackoverflow Data 32
4.1 Data Source 32
4.2 Data Dump Structure 33
4.3 Data Statistics 34
4.4 Other Data Sources 34

5 Analysis of the Problem 36
5.1 Assembling the Dataset 36
5.2 Framework 38
5.3 Feeding the Data into a Neural Network 38
5.3.1 Input Pipeline 38
5.3.2 Preprocessing 39
5.4 Approaches 40
5.4.1 Word Embedding 41
5.5 Architecture 42
5.5.1 Common Features 43
5.5.2 Word Summation 45
5.5.3 BiLSTM Encoder 45
5.5.4 BiLSTM Code Encoder 45

6 Results and Evaluation 50
6.1 Data Pipeline Construction 50
6.1.1 Indexing Data into Elasticsearch 50
6.1.2 Dataset Export 51
6.2 Experimental Setup 51
6.2.1 Embedding 52
6.2.2 Word Summation Model 52
6.2.3 BiLSTM Encoder 52
6.2.4 BiLSTM Code Encoder Model 53
6.3 Model Results Evaluation 54
6.3.1 Other experiments 56
6.3.2 Comparison to SNLI 57
6.4 Results Discussion 57

7 Conclusion 59

List of abbreviations 60

Bibliography **61**

A Script Documentation **65**

A.1 Used Libraries 65
A.2 Data 65
A.3 Dataset 66
A.4 Network 67
A.5 Word2Vec 68

B Elasticsearch and Kibana Examples **70**

1 Introduction

Natural language understanding is one of the fields of natural language processing (NLP), which is under ongoing research. This work tries to bring improvements into this field, targeting expert (scientific) domains with specific language. Appropriate examples of such domains can be found among the 173 Stackexchange's question and answer platforms. The topics covered range from computer science to biology and chemistry to personal finance. The individual domains require to target specific aspects of the users' language, which we demonstrate on the Stackoverflow page. However, general principles shown in this work can be applied to other domains as well.

The future usage of this work is to improve tasks such as information retrieval or code generation, which is beyond the scope of this work. In the case of the firstly mentioned, the vector representations of questions obtained by proposed neural networks can be used to search for the same questions already asked and answered. Therefore, it would be possible to get the correct answer immediately after describing the problem. Such application of the gained representations shall outperform basic keyword-based approaches that can result in very misleading results. Furthermore, this can also be used for automatic duplicate question detection on question and answer platforms.

An approach of this work is to use marked duplicate questions from the Stackoverflow to train a neural network to classify whether two questions are duplicates in terms of their semantics. Given this task, hidden layers of the neural network are forced to generate vector representations of the fed in questions, which are the desired outcome. An essential aspect of this work is also to utilize the information contained in code snippets present in the questions.

This bachelor thesis is structured as follows. In the first part, basic concepts and techniques of neural networks and natural language processing are briefly described. Later in the second part, results and realization of the given targets are going to be presented in detail.

2 Neural Networks

Neural networks can be defined as a subset of machine learning algorithms and can be utilized for supervised, unsupervised, as well as reinforcement learning tasks [1]. The neural networks use simple computational units (artificial neurons) organized into interconnected layers that form a computational graph. This concept, firstly introduced in the 1950s, is loosely inspired by a human brain [2] and is often used for various tasks [3] such as:

- image classification
- speech recognition
- handwriting transcription
- machine translation
- autonomous driving
- natural language understanding

...

The utilization of the neural networks provides nearly human-level performance for many of the tasks above. Moreover, there are already tasks (e.g., playing a game called Go) where this kind of artificial intelligence outperforms humans.

The following sections describe basic concepts and building blocks of the neural networks. Mathematical details are often omitted due to their complexity and good coverage by the referenced literature.

2.1 Neural Network Architecture

The architecture of a neural network can be divided into three levels of abstraction. These are artificial neurons, layers and models, where each subsequent part consists of the previous ones. The following sections describe these building blocks of the most basic neural network type - a multilayer perceptron. The multilayer perceptron consists of fully connected (*dense*) layers organized in sequential order. The basic concepts introduced in this section mostly apply to other kinds of neural networks (section 2.7) as well.

2.1.1 Artificial Neuron

A cornerstone of a neural network is an artificial neuron. Each neuron has n inputs $x_1, x_2, x_3, ..., x_n$. Each i-th input has its associated weight ω_i, which is used for computing a weighted sum (expression 2.1).

$$\sum_{i=0}^{n} x_i w_i \tag{2.1}$$

Furthermore, there is a number b (called bias) associated with the neuron. To get the output y we add the bias to the weighted sum and pass it through a non-linear function, which is called an activation function (see section 2.2 for more details). The computation in a neuron is described by equation 2.2 [4]. The structure of a neuron is illustrated in figure 2.1.

$$y = \varphi(\sum_{i=0}^{n} x_i w_i + b) \tag{2.2}$$

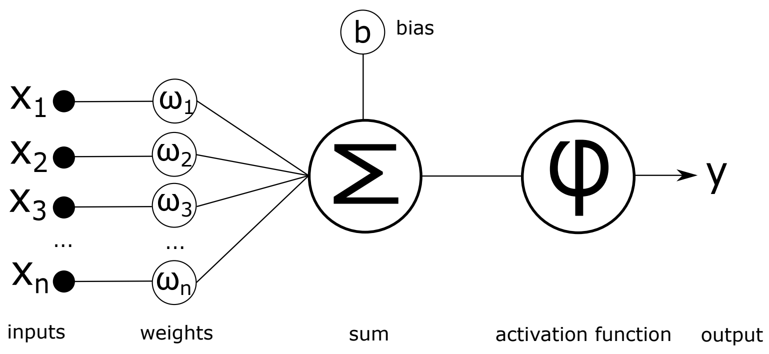


Figure 2.1: Symbolic structure of an artificial neuron.

2.1.2 Layer

Generally speaking, a layer represents an arbitrary transformation of input data inside the neural network. It is the essential building block of the neural network model in many deep learning frameworks (Tensorflow, Keras and et cetera). An organization of layers forms a computational graph defining operations performed on data. Layers do not even have to consist of any artificial neuron at all since a layer can be defined using other mathematical operation over the input data. However, the rest of this subsection is going to discuss a *dense* layer, which is made up of the neurons introduced in section 2.1.1.

When neurons are organized into the layer, it is not necessary to compute an output activation of each neuron one by one. Instead, the vector form of the equation for one neuron can be used to calculate the activation of the whole layer (equation 2.3). W^T then denotes a transposed matrix of all the weights in the layer, x is a vector of inputs and b is a vector of biases [4]. The function φ is the activation function and is applied element-wise on the resulting vector $W^T x + b$.

$$y = \varphi(W^T x + b) \tag{2.3}$$

Equation 2.4 shows a detailed expansion of equation 2.3 for a layer made up of two neurons and three inputs. In the expanded equation, y_i denotes an output activation of the i-th neuron, w_{ij} indicates a weight of the j-th input in the i-th neuron, b_i is a bias of the i-th neuron and finally, x_j represents a value of the j-th input [4].

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \varphi(\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}) \tag{2.4}$$

2.1.3 Model

The way how different layers are organized and connected (figure 2.2) is called a neural network model. Generally, a model consists of three parts - an input layer, hidden layers and an output layer. The input layer stands at the very beginning of the network and represents an entry point for data. If the input is an N-dimensional vector, then the input layer has N neurons inside. Moreover, the first layer is simplified and does not perform any computation. The only thing done is that the output activation of each neuron is set to the value of the corresponding input ($y_i = x_i$). Subsequent layers (*hidden layers*) perform the main part of the computation (section 2.3). Finally, activations in the output layer, which stands at the end of the model, are used to determine a resulting class (section 2.3).

2.2 Activation Functions

An essential part of each neural network is a non-linear activation function, which is used to obtain the activation of a neuron/layer (equation 2.2). Its most significant contribution is introducing a non-linearity into the model. Without such function, the neural network would be able to model only a linear dependency of a class on input data. It is because the output of

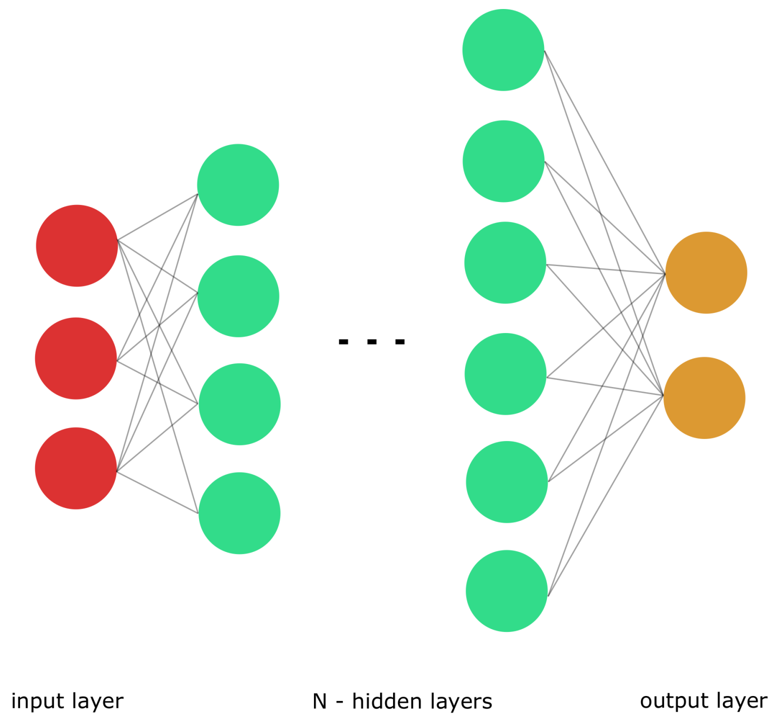


Figure 2.2: Example of a neural network model with four inputs, N hidden fully connected layers and two neurons at the output.

the model would then consist of a linear combination of the input and the network parameters only [3].

2.2.1 Sigmoid

A sigmoid function (equation 2.5) is commonly used in output layers for tasks such as logistic regression or multi-label classification tasks. However, it is suitable for hidden layers as well [5]. Its major drawback is a vanishing gradient (draws near zero) for low and high values of the z.

$$\varphi(z) = \frac{1}{1 + e^{-z}} \tag{2.5}$$

2.2.2 Hyperbolic Tangent (TanH)

A hyperbolic tangent (equation 2.6) is similar to the sigmoid function in its shape and drawbacks, with the difference that a range of values is $< -1, 1 >$, whereas sigmoid's range is $< 0, 1 >$. This activation function is predominantly used in hidden layers [6].

$$\varphi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.6}$$

2.2.3 Rectified Linear Unit (ReLU)

A rectified linear unit is a straightforward activation function, which is defined as a maximum of zero and z. A significant advantage of ReLU is that it is computationally efficient and despite its simplicity, it turns out to work very well for a wide range of tasks. It is typically used in hidden layers [5].

$$\varphi(z) = max(0, z) \tag{2.7}$$

2.2.4 Softmax

A softmax function (equation 2.8)[7] is frequently used in output layers of the neural network thanks to its ability to transform the inputs into a probability distribution for the classes. It means that a sum of the softmax's output vector elements is always equal to one. In combination with a cross-entropy loss, it forces the network to give the highest predicted probability to the class given by the label and to assign as low probability as possible to the rest of them.

It is worth mentioning that including the softmax in the activation functions section is questionable. It is due to the impossibility to compute an activation of a single neuron using the softmax. This fact can be seen in equation 2.8, where the knowledge of an inner product z_j (bias added to the weighted sum) of all the neurons in the layer is necessary to scale the inner product of the current (i-th) neuron z_i. Therefore the softmax is often perceived as a self-standing softmax layer.

$$\varphi(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.8}$$

2.3 Classification Using a Neural Network

A classification task can be defined as a process of feeding in one sample of data, passing it through the whole model and determining the resulting class. This process is called a forwardpass and starts with setting output activations of neurons in the input layer according to values of the input. Then, the activation of the input layer is used as an input for the subsequent hidden layer. Generally, activation of the n-th layer is used as an input for the $(n + 1)$-th layer. This step is done repeatedly until the output layer is reached.

Finally, when the activation of neurons in the output layer is known, a resulting class can be determined. Let us say that each neuron in the output layer represents one class, then a result of the classification is the one with the highest activation.

2.4 Loss/Cost Functions

As further explained in section 2.5, the loss function is used during training as a measure of how good or bad the current setting of network parameters is. It should be noted that the value of the loss is not a suitable measure of a network accuracy. For this purpose, metrics introduced in section 2.6 shall be used. Moreover, the selection of the loss function should be tailored to the metric we want to maximize, due to uncorrelation between some losses and metrics (section 2.4.3).

2.4.1 Mean Squared Error

Mean squared error (MSE) is a loss function that is defined by equation 2.9 and suits well for regression tasks [3]. In the equation, C denotes the loss function, w and b are sets of all weights and biases of the neural network, respectively and X is the current batch of inputs. On the right-hand side, there is a sum of the differences of labels y and predicted values $\hat{y}$ summed over all inputs in the batch. Finally, the sum is divided by the number of examples n.

$$C(w, b, X) = \frac{1}{n} \sum_{x} (y - \hat{y})^2 \tag{2.9}$$

2.4.2 Cross-Entropy

Another kind of a loss function is a cross-entropy loss (equation 2.10), which is often used for classification problems [3]. The left side of the equation is identical to equation 2.9 and is described in section 2.4.1. On the right side, there is a negative-sum of product $y_c log(p_c)$ over all possible classes M. In the product, y_c is a binary class indicator - equal to 1 if the true class of a current sample is c, else 0. Finally p_c is a probability of the current sample belonging to class c [8]. Value of p_c is determined by a softmax output layer (section 2.2.4) with which, the cross-entropy should be used.

$$C(w, b, X) = -\sum_{c=1}^{M} y_c log(p_c) \tag{2.10}$$

2.4.3 F1 Loss

In some cases, it might happen that the loss value does not correlate with a metric meant to be maximized. In other words, even if the loss decreases, the maximized metric may decrease as well. However, the expectation is that if we get better (lower) loss, we get a better (higher) value of the metric. This phenomenon can appear when the maximized metric is an f1 score (section 2.6.2) and the chosen loss is a cross-entropy [9]. In situations such as this, it is necessary to select another loss function, which can result in a better-trained model.

To maximize the f1 score (section 2.6.2) it can be favorable to use the f1 loss. Equation 2.11 states how the f1 loss is calculated. The equation is the same as in the case of the f1 score. However, the meaning of TP (true positives), FP (false negatives), FN (false negatives) is different from the f1 score. That is to be able to differentiate the loss. For example, if a label is 1 and a model's prediction is 0.8 probability for class 1, we count $TP = 0.8 * 1$, $FN = 0.2 * 1$ and $FP = 0.8 * 0$. Generally, this calculation is described by equations 2.12 [10].

$$C(w, b, X) = \frac{1}{n} \sum_{x} (\frac{2TP}{2TP + FP + FN}) \tag{2.11}$$

$$\begin{aligned} TP &= prediction * label \\ FN &= (1 - prediction) * label \\ FP &= prediction * (1 - label) \end{aligned} \tag{2.12}$$

2.5 Training a Neural Network Model

Thanks to equation 2.3 and a description of the forwardpass in section 2.3, it shall be clear that a result of classification is a function of input values, weights and biases. Since the input values are fixed, the only way how to affect the behavior of a neural network model is to adjust the weights and biases. The weights and biases are called parameters of the network.

The process of setting the weights and biases (which may be randomly initialized at the begging) to maximize the accuracy of the network is called training. To measure how optimal our current setting is, we define a loss function (section 2.4). A higher value of the loss means a worse accuracy of the network. In other words, we try to minimize the loss. However, the loss is a function of thousands of variables (mostly network parameters), which is the reason why it is not possible to find optimal values for all the parameters analytically. Due to that, an approximation of the loss minimum has to be found using gradient optimization methods (see section 2.5.1 for details).

The training procedure (minimizing the loss) is described in the following steps, which are performed in a loop over all dataset batches [3]:

1. Select a batch of training examples from a training dataset.
2. Do a forward pass with the selected examples and get predictions.
3. Compute the value of a loss over the current batch using the obtained predictions and true labels.
4. Compute an error of each neuron in an output layer using the loss.
5. Propagate the error to previous layers (this process is called a back-propagation).
6. Adjust parameter values to minimize the error (minimize the loss).

The loop over all the batches is performed multiple times (*epochs*). Since the previous steps are only a brief description of the training process, refer to the book by M. Nielsen [4] for further details.

2.5.1 Optimizer

An optimizer represents an exact algorithm that is used for adjusting the network parameters [3]. Generally, the optimization is based on computing a gradient (direction of the steepest increase) for each parameter with respect to the loss. The calculated gradient is used to move a value of the parameter against the gradient direction by a certain amount of the gradient's size. The amount of change is called a learning rate and is denoted as η. The learning rate is a member of a particular group of parameters called hyperparameters. They represent network parameters that are not learned during the training.

Examples of commonly used optimizers are listed below:

- Stochastic gradient descent (SGD) [11]
- RMSProp [12]
- AdaGrad [13]
- Adam [14]

2.6 Metrics

As stated in section 2.4, a loss function is not the best for measuring how well a network is trained. For this purpose, it is necessary to use other metrics that are presented in this section.

2.6.1 Accuracy

Accuracy is defined as a fraction of correctly classified examples (equation 2.13). Even though this metric is the most often used one, it is not suitable for datasets with an unbalanced number of samples in the classes. For instance, if there is a binary classification task and we have a dataset with 95% of "true" examples, the classifier can achieve a 95% accuracy only by classifying all the data as "true". It means that it might be reasonable to use another metric, such as an f1 score (section 2.6.2), that does not suffer from this drawback.

$$acc = \frac{correctly\ classified}{total\ number\ of\ examples} \tag{2.13}$$

2.6.2 Precision, Recall and F1 score

Before defining what a precision, recall and f1 score are, it is necessary to establish the concept of true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN), which are possible results of a binary classification. Table 2.1 states the individual cases and assigns a correct result class (TP, TN, FP, or FN) to them [5].

Using the previously declared result classes, the precision, recall and the f1 score can be defined. The precision (equation 2.14) is a fraction of correctly classified positive samples from all samples classified as positive (equation 2.15). The recall is a fraction of correctly classified positive samples from

Label	Prediction	Result class
1	1	TP
1	0	FN
0	1	FP
0	0	TN

Table 2.1: Possible results of binary classification and underlying result classes.

all the truly positive samples. Finally, the f1 score is defined as a harmonic mean of precision and recall (equation 2.16). As stated in section 2.6.1, the f1 score is extremely useful for the evaluation on unbalanced datasets.

$$precision = \frac{TP}{TP + FP} \tag{2.14}$$

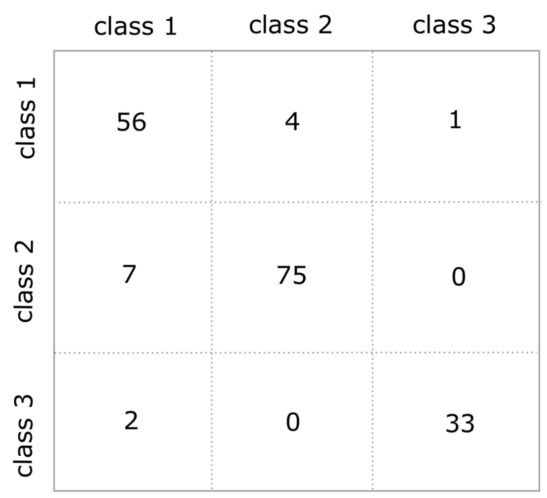
$$recall = \frac{TP}{TP + FN} \tag{2.15}$$

$$f1\_score = 2\,\frac{precission * recall}{precission + recall} \tag{2.16}$$

Even though the stated computation works for binary classification only, there are three possible ways of how to generalize the f1 score for multi-class classification. This generalization can be made using a micro, macro and weighted f1 score [15].

2.6.3 Confusion Matrix

A confusion matrix (figure 2.3) is a handy metric for evaluating network behavior in detail. Its significant benefit is that it shows which classes are being confused during the classification. It means that the confusion matrix is a $C \times C$ sized matrix where each member n_{ij} of the matrix denotes the number of cases when a sample of the i-th class was classified as the class j. Thanks to that, we can identify classes that the model struggles to distinguish the most. The confusion matrix may also appear in a normalized form, where each row is normalized to one.



	class 1	class 2	class 3
class 1	56	4	1
class 2	7	75	0
class 3	2	0	33

Figure 2.3: An example of a confusion matrix for three-class classification. For example, it shows that a sample from class 2 was classified as a class 1 for seven times.

2.7 Other Neural Network Concepts and Techniques

A concept of the multilayer perceptron does not suit very well for all tasks where neural networks can be utilized. Therefore, many other architectures dealing with different drawbacks of the original concept were developed. One such architecture, which tries to address a problem of multi layer perceptron's inability to deal with sequential data, is a recurrent neural network (RNN) [16] and its often used bi-directional form [16]. An extension of this concept is a gated recurrent unit (GRU) and a long short term memory (LSTM)[17] network that can handle long term dependencies in the sequences. The concept of RNNs is often amended by an attention mechanism [18].

Another type of architecture is a convolutional neural network (CNN), which applies a convolution operation over a sliding window to discover patterns emerging in the input [5]. This concept is often used for image or video classification and natural language processing.

Although these concepts are quite diverse, they all suffer from a common problem, which is overfitting [4]. This phenomenon can be identified by achieving perfect training accuracy while having poor results on a validation part of the dataset. To cut down the influence of the overfitting, regularization techniques can be used. An example of a regularization technique is an L2 regularization [4] or a dropout [19].

2.7.1 Siamese Neural Networks

The previously discussed multilayer perceptron can be perceived as a sequential network (figure 2.4), since layers are organized into an ordered sequence that defines the flow of computation. However, the sequential network that accepts one input vector (or one sequence) is not the only type of neural network architecture.

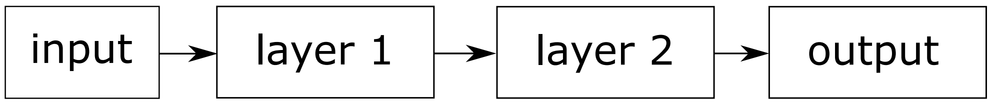


Figure 2.4: An example of a sequential neural network.

Generally, the neural networks can have multiple input vectors or sequences and the organization of the layers can be diverse. A special type of architecture is a siamese network [20]. This kind of network accepts two inputs that are passed through two parallel subnets sharing their weights. The two subnets can be followed by other layers processing an output of the subnets. An example of siamese network architecture is depicted in figure 2.5. Such architecture finds its usage, for example, in facial recognition or semantic similarity (section 3.3).

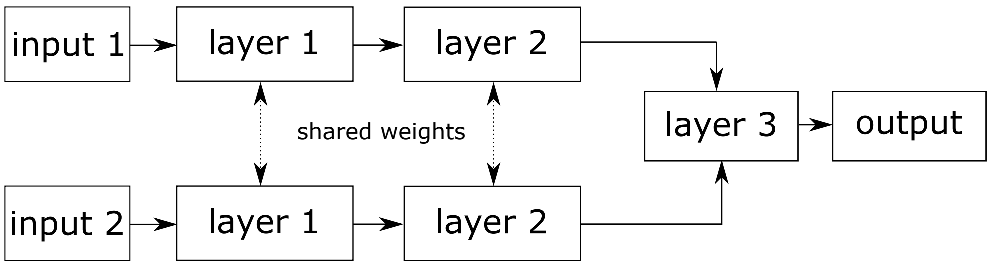


Figure 2.5: A siamese neural network that accepts two inputs and produces one output vector. The layers on the same level share their weights, as marked in the picture.

3 Semantic Vector Representation of Text

A natural language processing (NLP) is a subfield of linguistics and computer science. In recent years many deep learning techniques have been successfully applied to many NLP problems. These are, for instance, sentiment analysis, machine translation, text classification, semantic similarity and many more [21].

A problem in the NLP is that a dimension of the space of possible words or sentences fed into a neural network is immensely high. The way how neural networks deal with it is that each subsequent layer creates more abstract and usually lower-dimensional features from the one created by the previous layer. Figure 3.1 shows such a hierarchy, where the words are fed into a neural network as a one-hot representation. Then, subsequent layers create more abstract features, which can be perceived as word embeddings, sentence embeddings, higher-level features and finally, as a vector determining the resulting class.

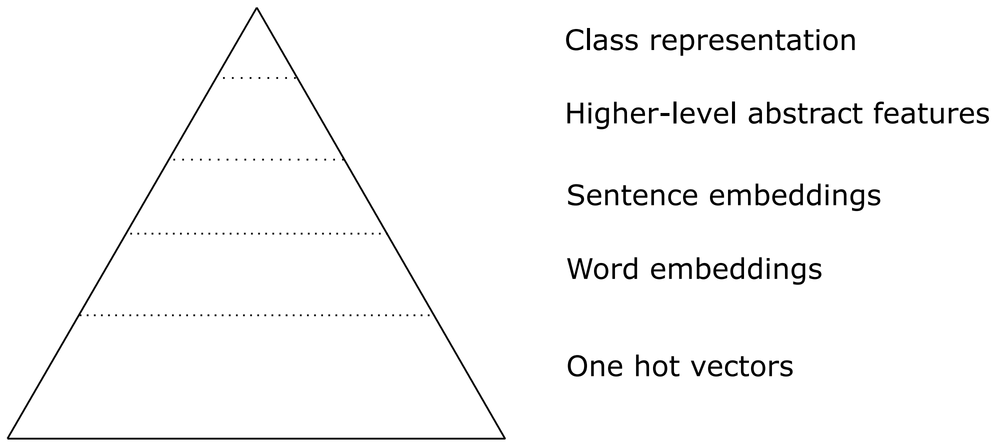


Figure 3.1: A pyramid of abstraction of processing a natural language using neural networks

In some cases, it might be beneficial to pre-train a part of the neural network in advance. For example, it is possible to pre-train word embeddings or a whole sentence encoder that is later used to build the final neural network. The pre-training can be done in a supervised as well as unsupervised way.

3.1 Vector Representation of Words

An embedding of words can be directly useful for a lot of NLP tasks. Moreover, since sentences are made up of words, text representation techniques are often based on the word embeddings. The desired word representation is a low dimensional vector (about hundreds of dimensions) of real numbers that captures the semantics of the word. Put differently, it is expected that two similar words appear close to each other in a vector space of the representations. A favorable fact is that the embeddings can be trained using unsupervised learning on a large text corpus. Such techniques are usually based on a distributional hypothesis [22].

In this section, two types of word representations - context-free and contextual representations are presented. At the end of the section, a concept of subword models is also briefly discussed.

3.1.1 Context-Free Word Representation

Context-free word representation techniques are identified by creating a fixed vector for each word, no matter its context. It means that, for example, the word "bank" would have the same representation when it appears in a context "bank account" as in case it appears in a context "river bank" [23].

An advantage of this approach is that unlike the contextual representations, the word embedding is self-standing. Therefore, a similarity score of two words can be computed and used for similar word search.

Word2Vec

A Word2Vec [24] is an unsupervised technique to represent words as vectors. Generally, this algorithm can be found in two forms. The first one is called a continuous bag of words (CBOW) and lies in predicting a target word given its context. The other method, on the contrary, predicts the context given a central word and is known as a skip-gram model.

The skip-gram model is trained in a way that the central word is used as an input to a neural network that produces a conditional probability of the central word co-occurring next to other adjacent words in a sliding window. The window size defines the number of neighboring words taken into account. Then the vector representing the central word (which is stored in a weight matrix of the network) is adjusted to maximize the computed conditional probabilities for the neighboring words. The way how the CBOW model is trained is analogical and further explained in [25] together with mathematical details of the skip-gram model.

Global Vectors (GloVe)

Global Vectors (GloVe) [26] are another method of creating word embeddings. Unlike Word2Vec, GloVe utilizes a global word co-occurrence statistic that is captured in a co-occurrence matrix. To turn such a matrix into a vector representation, the authors introduced a new algorithm. It is based on optimizing the word vectors so that the dot product of the two word embeddings is equal to a logarithm of their co-occurrence count.

3.1.2 Contextual Word Representation

As stated in section 3.1.1, the context-free representations do not capture the context in which a word is used and this feature may be undesirable in many NLP tasks. Fortunately, contextual word representation techniques exist. The idea is that hidden layers in an LSTM based language model contain contextualized representations of fed-in words. The usage of such embeddings is then different than with the fixed representations, such as pre-trained Word2Vec (section 3.1.1). Instead of querying a lookup table, the vector embedding is obtained by passing the words into a model that produces the contextual embedding later used as an input to a task-specific neural network. An example of contextual word representations based on the deep bi-directional LSTM language model is ELMo (Embeddings from Language Models) [27].

Another contextual embedding method is a BERT [28] (Bidirectional Encoder Representations from Transformer) model. This model is based on Transformer architecture [29], and its utilization provides outstanding results in many tasks.

3.1.3 Subword Embedding

A significant drawback of the techniques presented in section 3.1.1 is that they are not able to handle uncommon and out of vocabulary words (OOVs) - word for which an embedding does not exist. Techniques trying to overcome this obstacle are called subword embeddings and are based on dividing words into smaller units down to a character level. For instance, the word "inevitable" can be composed of two units "in" and "evitable".

Byte-Pair Encoding

A byte-Pair encoding [30] is an unsupervised text segmentation algorithm which originated as a compression method. The algorithm starts with a list of elementary symbols, such as characters. Then it creates a new symbol composed of the most frequent symbol pair that appears in the training text. This step is repeated until the desired size of the vocabulary is achieved.

WordPiece

Another unsupervised method for subword tokenization is a WordPiece algorithm [30], which is to some extent similar to the Byte-Pair encoding (section 3.1.3). The difference is in the merging step. Instead of combining the two most frequent pairs, WordPiece chooses a couple, that would increase the log-likelihood of a language model [22] if added to the vocabulary.

3.2 Vector Representation of Sentences

As stated in the introduction of chapter 3, having a robust vector representation of text may be advantageous or even necessary for many NLP tasks. In this section, some of the used methods for obtaining semantic sentence embeddings are presented.

3.2.1 Combining Word Embeddings

Simple methods obtain a vector representation of text by combining pretrained word vectors. One way how to combine the word embeddings is a summation or a weighted average, where the weights are tf-idf scores computed from a language corpus [31]. The tf-idf score for the i-th word is calculated according to equation 3.1, where k is the number of all words in the j-th document and n_{xy} denotes an occurrence count of the x-th word in the y-th document. Finally $|D|$ is the number of documents in the corpora and $|\{j : t_i \in d_j\}|$ is the number of documents where the word i appears.

$$tf\_idf_i = \frac{n_{ij}}{\sum_k n_{kj}} * log\frac{|D|}{|\{j : t_i \in d_j\}|} \tag{3.1}$$

3.2.2 Supervised Representation Learning

A different way how to get a robust vector representation of text is to train a neural network model on a supervised learning task since the sentence embedding would then lie in hidden layers of the neural network. For example, in case of a network that consists of an embedding layer on the beginning, two LSTMs above it and two dense layers at the output, the representation of the sentence can be taken from the output of the second LSTM layer. A significant downturn of this method is a necessity of a labeled dataset that may not be available in many cases.

The quality of the embeddings obtained by this technique depends on the choice of a learning objective. Examples of such tasks are, for instance, machine translation or natural language entailment (section 3.3.1). Another supervised learning objective is a semantic similarity task, which aims to recognize whether two texts are similar or not on a semantic basis. This task is going to be further focused by the rest of the work and a more detailed description can be found in section 3.3.

3.2.3 Unsupervised Representation Learning

In cases when a labeled dataset is not available, an unsupervised representation learning might come into play. Generally, a principle of the supervised representation learning (section 3.2.2) applies for the unsupervised case in the same way - the desired vector representation of sentences is in the hidden layers. The difference is the choice of the learning task, which is unsupervised in this case.

To give an example, the BERT model [28] is trained on two unsupervised tasks. The first one is to predict a randomly masked token from an input sequence based on its context. The second one is the next sentence task, which aims to predict whether one sentence follows the other one in the original text. Another unsupervised learning model called GPT-2 [32] is trained on predicting the next word in a text.

3.3 Semantic Similarity Task

In many real applications of the NLP, such as search engines, it is essential to measure or identify whether two texts are similar or not. Although this may look easy to accomplish just by looking if both documents contain the same keywords, distinguishing between tiny semantic nuances may be a severe problem. To differentiate texts with minor semantic differences is a target of a semantic similarity task, which is under ongoing research at the moment. Generally, the approach is to embed sentences into an n-

dimensional space and then to compute a cosine similarity (equation 3.2) or another metric of the two vectors.

$$similarity = cos(\theta) = \frac{A \cdot B}{||A||\,||B||} \tag{3.2}$$

Therefore, the main objective of the semantic similarity task is to create embedding vectors that contain robust semantic information (section 3.2). The following sections present the related datasets and techniques.

3.3.1 Related Datasets

For the semantic similarity task, there is not a lot of data available, which is a crucial problem in this field. While some datasets suffer from a small size, some of the others are partially made up of automatically generated samples, which may be undesirable as well. In this section, some of the available datasets are discussed.

STS Benchmark

STS benchmark dataset [33] is a subset of data used in SemEval competitions. Examples contained in the dataset come from various sources such as user forums, image captions and news headlines. The dataset is separated into three subsets - training, development and testing. The total size of the dataset is around 8 600 samples and the sizes of the dataset parts are shown in table 3.1.

Dataset part	**Dataset size**
training	5 749
development	1 500
testing	1 379

Table 3.1: Sizes of training, development and testing parts of the STS Benchmark dataset.

Stanford Natural Language Inference Dataset

A Stanford Natural Language Inference dataset (SNLI) [34] consists of human-labeled pairs of sentences. The possible labels are a contradiction, entailment, and neutral relationship. Although a learning objective of this dataset is not the semantic similarity, a natural language inference task requires a meaningful vector representation of sentences in the same way as the semantic similarity does. In other words, the significant difference lies in the classifier built over the representations. Thanks to that, the SNLI can be used for both an evaluation and training of the semantic representations of a text.

The SNLI dataset, which consists of 570 000 examples, is split into three parts - training, development and testing. The sizes of these parts are shown in table 3.2. The amount of data and a lack of automatically generated examples make SNLI a very reasonable choice for many researchers. An example of the dataset pairs with the corresponding labels from the original paper [34] are shown in table 3.3.

Dataset part	**Dataset size**
training	550 152
development	10 000
testing	10 000

Table 3.2: Sizes of training, development and testing parts of the SNLI dataset.

Text	**Hypothesis**	**Label**
A man inspects the uniform of a figure in some East Asian country.	The man is sleeping.	contradiction
An older and younger man smiling.	Two men are smiling and laughing at the cats playing on the floor.	neutral
A soccer game with multiple males playing.	Some men are playing a sport.	entailment

Table 3.3: Three examples from the SNLI dataset.

3.3.2 Related Work

Since the rest of the work focuses on the vector representation of text enriched with source code snippets, the following section presents an existing work on related topics. The paragraphs provide a brief description of the papers. A more detailed description can be found in the cited sources.

A large annotated corpus for learning natural language inference

Authors of the SNLI dataset published the results of three different models in their paper [34]. Generally, all of their models consist of a word embedding layer for both premise and hypothesis and a sentence embedding layers on the top of the word embeddings. In the end, classification is done by three 200-dimensional dense layers taking the 100-dimensional representation of the premise and hypothesis as an input. At the output, there is a softmax layer. A difference between these models is a method of sentence representation. The first and most simple model uses a sum of the word embeddings. The next one uses a simple RNN layer, whereas the last one uses an LSTM layer for sentence representation. The latest mentioned model is the most successful one with a test accuracy of 77.6%.

Shortcut-Stacked Sentence Encoders for Multi-Domain Inference

In 2017, Nie X. and Bansal M. presented their work on a Shortcut-Stacked Sentence Encoder [35]. It aims to represent a sentence as a vector of real numbers that is later used for an NLI classification. The model is made up of three bi-directional LSTM layers with shortcut connections and a classifier on top of it. The shortcut connections are made in the way that input of each LSTM layer consists of an original word embedding concatenated with a corresponding sequential output of all the preceding LSTMs. Such an approach led to an 86.0% accuracy on the SNLI dataset.

Sentence Embeddings in NLI with Iterative Refinement Encoders

Another work [36] related to obtaining a vector representation of text introduces a hierarchical structure of bi-directional LSTM layers that implements an iterative reinforcement strategy. The key idea behind the model is that each subsequent LSTM layer is initialized with the last hidden state of the preceding layer while accepting the original word embeddings as an input. An output of each LSTM is then max-pooled and concatenated to a resulting representation. The authors have reported a very good accuracy of 86.6% on the SNLI dataset.

Dynamic Self-Attention Model

Yoon D., Lee D. and Lee S. in their paper [37] presented a new Dynamic Self-Attention (DSA) model, which outperforms many related models. DSA is based on attending to a dynamic weight vector rather than to a fixed vector learned during training. The model succeeded on the SNLI dataset with a result of 87.4%, which is the highest score among reported results in a sentence vector-based group of models.

Semantics-aware BERT for Language Understanding

At the moment, the best reported result on the SNLI dataset is achieved by a model called Semantic-aware BERT (SemBERT) [38]. It is a BERT [28] model extended by information about a semantic role of different parts of an input. More precisely, the BERT model is used to compute a contextual representation of word tokens. It is then concatenated with a vector, which carries the information about the semantic role of the token in a particular sentence. An achieved accuracy on the SNLI dataset is 91.9%.

Retrieval on Source Code: A Neural Code Search

In their paper, Sachdev S. et al. [39] addresses a natural language search over a large code base acquired from the GitHub. They use a variant of a Word2Vec embedding (section 3.1.1) to obtain a representation of the query and a code snippet. Three methods of creating a document embedding based on the word representations are evaluated. These are simple averaging of all the word vectors, averaging unique words only and a weighted average using the tf-idf (equation 3.1). Such embedding of the query is then used to find a code snippet with its representation closest to the query embedding. The accuracy of the code search was evaluated on 100 hand-picked StackOverflow questions with an accepted answer. The most promising result is achieved using the weighted average method, where an acceptable solution was found for 43 out of 100 test queries.

When Deep Learning Met Code Search

Another existing work [40] extends the previously discussed method by introducing supervision into the training. A significant difference is that the authors use two different embedding matrices (the first one is used for the code and the second for the query), initialized by the same pre-trained weights. These matrices are then fine-tuned during the training process. Additionally, the scheme of producing a document level embedding from the word token embeddings is changed. Instead of the tf-idf weighted average, a simple average is used for the query and a weighted average based on learned attention weights is used for the code.

4 Stackoverflow Data

A Stackoverflow is one of Stackexchange's community platforms, which is designated for programmers. It provides a space to ask questions and get answers from other members of the community. At the beginning of July 2019, the page contained more than 45 million questions and answers, which makes the Stackoverflow being considered as the biggest programming discussion platform.

Not only does the Stackoverflow store an enormous knowledge base, but the site also hides valuable hand-tagged data that can be used for machine learning. More precisely, users can mark two questions as duplicates so that it warns the others that there may be a desired solution already available. However, the interesting part thereof are the duplicate links themself. Thanks to them, the Stack Overflow data can be used to create a dataset for the semantic similarity task (section 3.3). Utilization of the duplicates for learning a sentence encoding is an aim of this work and will be focused in the subsequent chapters.

This chapter is structured as follows. Firstly a detailed description of the data source is given. Later the chapter discusses the structure of the data dump and presents essential data statistics. Finally, a short discussion about alternative data sources takes its place.

4.1 Data Source

The export of the complete Stackoverflow website is available at `www.archive.org/details/stackexchange`. The work uses a page dump from the beginning of July 2019. Therefore the information stated by this work is related to this date. The export comes logically separated in eight compressed XML files, each carrying different information. The individual parts and their sizes (compressed) are listed below:

- badges (242.7 MB) - gained honors
- comments (4.2 GB) - user comments
- post history (25.0 GB) - history of posts
- **post links** (84.7 MB) - relationship links between the posts

- **posts** (14.3 GB) - all questions and answers
- tags (797.9 KB) - tags that can be associated with the posts
- users (504.8 MB) - profiles of page users
- votes (1.1 GB) - votes for the posts

Only the post links and posts (both outlined using a bold font in the listing) thereof are used for assembling the dataset. A detailed description of the export parts relevant to the dataset is provided in the following section.

4.2 Data Dump Structure

As stated in the previous section, the Stackoverflow dump comes in XML files. Since not all the fields available in the XML are necessary, only the relevant ones are chosen to be further processed. A field listing of the posts and post links exports with the selected fields highlighted can be found below:

```
posts.xml
        - Id
        - PostTypeId
                -> 1: Question
                -> 2: Answer
        - ParentID (only present if PostTypeId is 2)
        - AcceptedAnswerId (only present if PostTypeId is 1)
        - CreationDate
        - Score
        - ViewCount
        - Body
        - OwnerUserId
        - LastEditorUserId
        - LastEditorDisplayName
        - LastEditDate
        - LastActivityDate
        - CommunityOwnedDate
        - ClosedDate
        - Title
        - Tags
        - AnswerCount
        - CommentCount
        - FavoriteCounts
```

```
postlinks.xml
        - Id
        - CreationDate
        - PostId
        - RelatedPostId
        - PostLinkTypeId
                -> 1: Linked
                -> 3: Duplicate
```

From the listing above, it can be seen that the linking of the duplicates is done using a post link record with $PostLinkTypeID = 3$. Each post link has its unique identifier (*Id*) and is assigned to one of the related posts via a unique identifier of the post (*PostId*). The second post in the relationship is also determined by its identifier (*RelatedPostId*).

From the post attributes, the most important fields are *Id*, *PostTypeId*, *Body* and the *Title*. Worth mentioning is that the *Body* attribute contains a formatted content of the post in an HTML.

4.3 Data Statistics

Table 4.1 shows document counts in different categories of the data. The first four of them are basic categories defined by a separation of the data source. The last four of them (separated by a horizontal line) are derived categories that are subsets of the posts.

From the table, it can be seen that the overall number of posts is around 45 million. 17.8 million of thereof are the questions, which are the point of interest in this work. Moreover, around 491 thousands of pairs of questions are the duplicates that will form a basis of the entire dataset. Another significant property observed in the data is that 76.6% of the questions contain a code snippet (body of the post contains an HTML tag "<code>").

4.4 Other Data Sources

Besides the Stackoverflow duplicate questions, other data sources exist as well. One of the alternative data sources can be even obtained from the Stackoverflow dump by extracting pairs made up of questions and their corresponding accepted answers. The idea behind that is to use the accepted answers to predict whether the post is an answer to the given question.

Dataset part	**Number of samples**
comments	74 003 667
users	10 640 388
post links	5 600 831
posts	45 069 473
questions	17 786 242
questions containing a code snippet	13 628 089
questions with an accepted answer	9 362 222
duplicate pairs of questions	491 337

Table 4.1: Example counts in different document categories in the Stackoverflow data dump. Below the horizontal divider are derived categories that are subsets of the "posts" category.

Alternatively, the size of the current dataset might be enlarged with data from the remaining 173 webpages of the Stackexchange platform. That would bring more complexity into the dataset since each page focuses on a different topic such as 3D printing or math. Furthermore, variants of few Stackexchange pages exist in languages other than English.

Apart from the Stackexchange, there are other similar web pages, such as Quora. That page was already utilized to create a Quora Question Pairs dataset consisting of approximately 404 thousands of training examples. A training objective of the dataset is to predict whether two questions are duplicates or not. The dataset can be accessed on `https://www.kaggle.com/c/quora-question-pairs`.

5 Analysis of the Problem

This work aims to create a sentence encoder using the dataset from Stackoverflow duplicate questions (chapter 4). Therefore, this work utilizes deep learning techniques to train a classifier distinguishing between different, similar (only some of the presented models) or duplicate questions. This chapter presents possible solutions for different sub-problems and discusses the chosen approaches.

5.1 Assembling the Dataset

The first step in training a neural network is to obtain a dataset. To solve the given task, this work chooses to utilize duplicate questions obtained from the Stackoverflow webpage (chapter 4). The Stackoveflow page was chosen from the Stackexchange pages since it is the largest one. Additionally, it requires to address the domain specific language containing code snippets.

To assemble the dataset, it is essential to clean the data source. In this case, the data contain duplicate links pointing to a not existing post (the previously stated number of duplicates is after cleaning the invalid links). That can be achieved by iterating over all the links and querying referenced posts. If one of the referenced posts cannot be found, the corresponding link is removed. All the remaining links are then used as a base for the dataset.

Generally, the learning objective of the created dataset shall be to classify whether two questions are duplicates or not. It implies that each example in the dataset is a triplet "(*master post*, *related post*, *class*)", where the class denotes the relationship between the two posts. Since the neural network must distinguish a tiny semantic difference in the questions, the dataset shall not consist of duplicates and different question pairs only. Therefore, the dataset must embrace similar question pairs as well. This work defines a question pair as similar if there is no duplicate link between the questions, but a keyword-based search query finds a match between them. Consequently, the dataset is made up of pairs of questions classified into the following categories: *different*, *similar* and *duplicates*.

To assemble the dataset, firstly, the duplicates are found and assigned to each other. The process is to iterate over all the duplicate links and query the first post of the current link. This post is marked as a *master post*. After that, all duplicates for the given *master post* that are not already in the dataset are found. During the duplicate search, the links are handled transitively up to a depth of five links. Handling the links transitively may result in marking two questions as duplicates even though they are different (in case of the transitive links of a higher order). However, table 5.1 shows that transitive duplicates of higher order are not represented in the dataset in significant numbers. All the found duplicates are the assigned to the *master post* and they are marked as *duplicates*.

Relation type	**Total examples**	**Percentage**
direct	222 656	98.48%
1st transitive	3 226	1.43%
2nd transitive	133	0.06%
3rd transitive	28	0.01%
4th transitive	25	0.01%
5th transitive	22	0.01%

Table 5.1: Summary of duplicate examples obtained from direct links and transitive relations.

After that, three similar posts that are not already in use shall be found for each *master post* in the dataset. To perform such queries over the data, they must be stored in a database providing such functionality. Currently, there are many software tools for data storage, analysis and search available on the market. Examples of such databases are, for example, MySQL, MongoDB, Elastic stack, et cetera. This work chooses to work with the Elastic stack (Elasticsearch + Kibana) since it comes in a free, open-source version and provides suitable functions for data indexing, searching, exploration and visualization. The posts found using the Elasticsearch are assigned to the *master post* and marked as *similar*.

In the end, three randomly chosen posts (that are not in use already) are assigned to each of the master posts as a *different* post. The different posts are chosen randomly since there is little likelihood that the post would be similar.

5.2 Framework

One of the essential decisions when it comes to neural networks is a framework choice. These days, many machine learning frameworks are available and each has its advantages and disadvantages. These are, for example, PyTorch, Scikit-learn, Tensorflow (TF), or Keras.

This work chooses to work with the Tensorflow 2.0 in combination with the in-built Keras. Together they provide extensive options to build, train and evaluate a neural network. Other significant advantages of the TF are the detailed tutorials that are freely available. Additionally, there is an active community that stands behind the TF. Moreover, a web application called Tensorboard can be used to display a computational graph of the network, view the training and evaluation results, or analyze how does a hyperparameter setup affects the resulting accuracy.

5.3 Feeding the Data into a Neural Network

Feeding dataset examples into a neural network for a purpose of training or evaluation requires to build an input pipeline. Its role is to load, shuffle and batch the examples. Furthermore, the raw data needs to be preprocessed before being ingested by the network. These points are discussed in detail in this section.

5.3.1 Input Pipeline

The design of an input pipeline heavily depends on the features provided by the chosen framework. The Tensorflow provides a wide range of possibilities of feeding in the data. The first one is to build a complete instance of a TF Dataset class, which provides functionality to download, split and feed the data into the network. However, this approach is quite labor-intensive. That may be undesirable in the early phases of development since there is a chance that there will be additional changes in the dataset.

Another feasible approach is to omit the Dataset API entirely and to use pure Python generators. Unfortunately, this results in losing a lot of useful functions that the API provides. A good compromise between implementation complexity and the features provided by the TF is to create an instance of a Dataset class from an external data source. The external sources that can be used are NumPy arrays, Python generators, TFRecords, CSV, et cetera. For development purposes on our dataset, two pipeline versions based on Python generators might be used. These two approaches are further discussed below.

Elasticsearch Generator

The first possible input pipeline works with an export of the dataset that consists of IDs of two posts and a corresponding label. The generator reads CSV lines and dynamically loads both posts from the Elasticsearch instance where the posts are stored. Then, it runs a preprocessing (section 5.3.2) and yields a dataset item. Unfortunately, it turns out that this type of generator is slow due to the need to download and preprocess the data on the fly.

Text CSV Generator

The second input pipeline utilizes another version of the dataset export that contains the text of the posts instead of their IDs. Such an approach provides better speed and the possibility to preprocess (section 5.3.2) the content in advance when exporting the dataset. Generally, the generator accepts one or two CSV files, depending on the model type. One of the files carries preprocessed text pairs and the other optional one carries a preprocessed code. The generator then reads the input file(s) and yields dataset examples.

5.3.2 Preprocessing

As stated in chapter 4, post content is an HTML code that carries formatting information for the webpage. Therefore, it is necessary to extract text from the HTML and to do other preprocessing steps described in the subsequent paragraphs. Furthermore, since a significant part of all the posts contains code snippets, the text and code are preprocessed separately. The preprocessing is done during an extended export of the dataset to save computational power during the training.

Text Preprocessing

The text is preprocessed in a few steps. Firstly new line characters "\n" are removed from the text. Then, all URL addresses are replaced with a special reserved token "<url>" and HTML tags are stripped while replacing a content of "<pre><code>" tags with a token "<code>". After that, date and time data are replaced with a reserved token "<datetime>", numbers are replaced with "<numbers>" and finally, special characters are removed from the text.

Code Preprocessing

The code preprocessing is slightly different from text preprocessing. Firstly, the content of all "<pre><code>" tags is extracted. Then, comments are removed from the source code and float and integer numbers are replaced with "<float_token>" and "<integer_token>" respectively. In the end, the new line characters are removed from the resulting code.

5.4 Approaches

There are many feasible approaches for classifying pairs of the Stackoverflow questions into categories expressing the measure of similarity. In this section, possible and chosen approaches are briefly discussed.

One possibility for classifying whole text is to combine the word embeddings of words as described in section 3.2.1. The resulting feature vectors can be passed through dense layers with a softmax output layer to determine th class. A significant advantage of this model is its simplicity, despite which it provides meaningful results. That is the reason why this model is often used as a baseline for datasets, such as the SNLI (section 3.3.1). To be able to evaluate results improvement of models proposed later in this work, the summation of word embeddings is used as a baseline model.

Another approach is to use an RNN encoder to produce a sentence representation based on fed in words or characters. RNNs and their variations turn out to perform very well (section 3.3.2) on many NLP tasks, which is the reason why other proposed models are based on the LSTM layers.

Recently 1D CNNs are also being utilized for text processing. Generally, the words or characters represented as vectors are passed through one or more 1D convolution operations, which produces the resulting vector representation of the sentence. The vector is then used for final classification in dense layers. The process of 1D convolution with one convolution filter is depicted in figure 5.1. An application of the CNNs to this problem is not examined in this work since the recent results on similar tasks show that the CNNs does not outperform RNN based solutions.

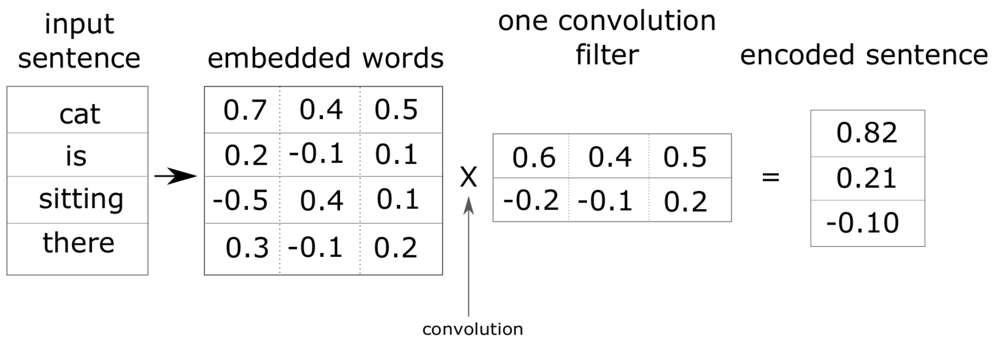


Figure 5.1: Sentence encoding using a 1D convolution with one filter.

Last but not least, there is a possibility to use pre-trained models such as BERT or GPT-2. These are powerful models trained for a couple of days on large clusters and are usually only fine-tuned during training on a final task. In other words, it is not feasible to train them on a custom dictionary from scratch in this work. That is a major drawback for the Stackoverflow dataset due to the specific dictionary of the users. Moreover, the posts contain code snippets for which neither the BERT nor GPT-2 is pre-trained.

5.4.1 Word Embedding

A lot of neural network models dealing with similar problems use pre-trained word embeddings. The reason is either to save a training time or to cope with a lack of labeled data. This work also chooses to utilize pre-trained embeddings for both textual parts of the posts and parts containing code. The reason is mainly to speed up the training.

The major questions regarding word representations are which embedding technique to use and how to treat the code. One approach that can be seen in work from Facebook [39] is to use a Fasttext (a variant of the Word2Vec) and to remove all special characters and programming language reserved tokens from the code. The result is a sequence made up of identifiers only.

This approach does not seem to be beneficial for this work since the mentioned preprocessing of the code results in losing information about the programming language. Moreover, it relies on using identifiers with reasonable names. For this work, it is essential to preserve the information about the used programming language, since this may be the major difference between the two questions. Similarly, the proposed models shall not rely on the usage of meaningful identifiers. This is because the questions may refer to a general feature of the programming language and the code snippet might not be supposed to do a meaningful activity.

These facts lead to choosing a different approach for code embedding, where the reserved words and special characters are processed by the network as well. Therefore, the *k* most common tokens in the complete dictionary (including tokens such as "{", "}" or "public") are used to pre-train their embeddings and the rest of them is handled as OOVs. It can be assumed that all the OOVs that appear in the code are variable or function identifiers.

Another important decision is the way how to treat OOVs. To handle them, this work creates a special "<OOV>" token, whose representation is learned during the end to end training, while the rest of the embeddings are pre-trained and fixed. Additionally, representations of other special tokens described in section 5.3.2 are pre-trained as well. Other possible ways of how to handle the OOVs are subword models (section 3.1.3), character level embedding, or a mixture of the word and character level embedding.

To pre-train the embeddings, this work chooses to work with the Word2Vec - CBOW model, which is a frequent choice of many related works. Moreover, there is a pre-trained Word2Vec model available on a Tensorflow Hub, which can be used for a comparison with the custom trained embeddings.

5.5 Architecture

Generally, the proposed models use siamese neural network architecture (section 5.2), which is illustrated in figure 5.2. The siamese neural network in the picture accepts two questions, applies pre-trained embeddings and passes them through subsequent layers that share their weights between both branches. That results in generating a vector representation of both questions. These vectors are merged and passed through a multi-layer perceptron with a softmax layer at the end.

The most remarkable difference between proposed models is a sentence encoder, while the general architecture is the same. Other minor differences can be found in a softmax classifier part as well. These can be, for example, a number of layers or a number of neurons in the layers.

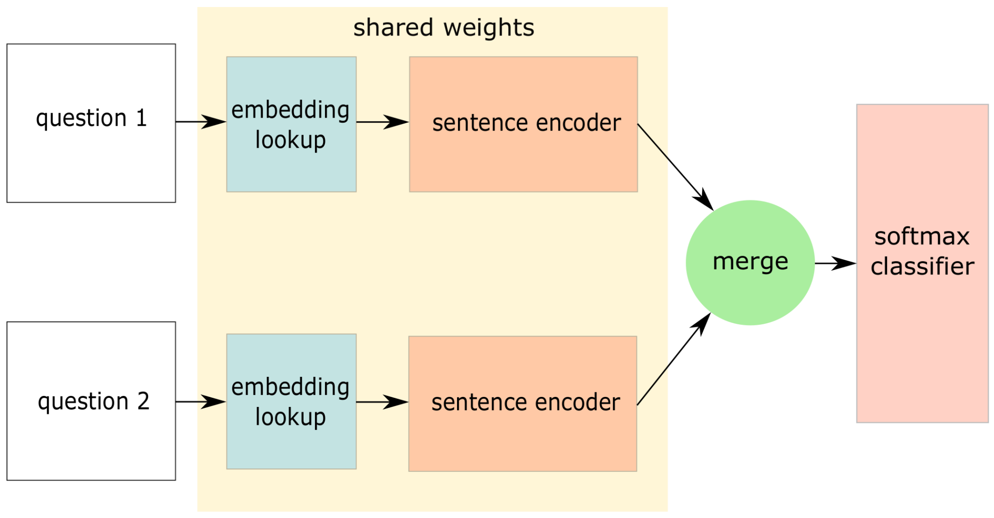


Figure 5.2: Siamese neural network architecture for classification on the Stackoverflow dataset.

The subsequent sections firstly introduce common features of all the models and then the individual models are described in detail.

5.5.1 Common Features

All the proposed models share common architectural elements such as an input pipeline, embedding layer, the way how sentence vectors are merged, metrics and loss. These common features are discussed in this section.

Data input

The question pairs are fed into the model in batches of a parametrizable size. An input pipeline takes care of loading the data, padding the sentences to the same length, shuffling and batching. The yielded batch contains pairs of sequences created from dictionary indexes of each word. Put differently, the input is a tensor of shape *(2, batch size, max sequence length)*.

Embedding Lookup

The first layer of each model is an embedding layer (*tf.keras.layers.Embedding*), whose weight matrix is initialized with the pre-trained embedding weights. The embedding layer creates a matrix made up of one-hot vectors that correspond to the dictionary indexes of fed-in words. The matrix is then multiplied with the weight matrix, which results in a tensor with shape (*batch* *size, max sequence length, embedding dimension*) for each of the input questions. A way how the OOVs are handled is similar for all the models and is described in section 5.4.1.

Merge Step

Another common feature of all the models is the merging of sentence representations. All proposed models work with a concatenation of the sentence vectors, which is also used in the SNLI dataset paper [34].

Optimized Metric

The choice of a metric (section 2.6) to be optimized and observed is very important. This work aims to optimize an f1 score since table 6.1 shows that the dataset is unbalanced. It means that accuracy would not provide interpretable results. Besides, a confusion matrix is used to provide more in-depth insight into the model's behavior.

Loss Function

An originally used loss function was a cross-entropy (section 2.4.2), which is a frequent choice of classification tasks with a softmax output layer. However, experiments showed that the cross-entropy does not correlate with the f1 score very well, causing worse results. To address this problem, an f1 loss (section 2.4.3) is used for training the models and is optimized using an Adam optimizer (section 2.5.1).

Regularization

A regularization (section 2.7) is an integral part of a neural network. The models proposed in this work use an L2 regularization (*tf.keras.regularizers.l2*) in all the layers of the softmax classifier. A regularization parameter is always the same for all the layers.

Additionally, the models utilize dropout layers (*tf.keras.layers.Dropout*). In the models, two different dropout configurations are used. The first one (which is usually configured to a higher dropout rate) is always after the embedding layer. The second dropout follows each of the subsequent layers.

5.5.2 Word Summation

The first proposed model (figure 5.3) is based on a word summation method described in section 3.2.1. It means that the whole sentence encoder is just a sum of the word embeddings. Encoded questions are then concatenated and passed through a dense layer and a softmax layer at the output. This model uses only a textual part of the dataset while omitting code tokens entirely (except the special "<code>" replacement token in the text).

This model is used in two variants that differ in a number of classes. A two-class model is a special case of a three-class model, where all the "similar" samples are treated as "different".

Variant	**Classes**
WordSum2Cls	2
WordSum3Cls	3

Table 5.2: Variants of a word summation model.

5.5.3 BiLSTM Encoder

The second proposed model is based on a bidirectional LSTM encoder and utilizes only the textual part of the dataset. Figure 5.4 shows one variant of the model, where the embedded sequences are passed through two BiLSTM layers that produce the vector representation of the sentences. The resulting representations are concatenated and used for classification in the softmax classifier.

Variants of the model differ in a number of classes, a number of BiLSTM layers in the encoder and a number of the dense layers in the softmax classifier. These variants are described in table 5.3.

5.5.4 BiLSTM Code Encoder

The last proposed model (figure 5.5) is different from the previous one in a way how the code parts of the dataset are handled since this model also utilizes the second part (with code tokens) of the dataset. It means that the model accepts four sequences - two of them carry the textual tokens and the others carry the code tokens. The textual sequences are then embedded

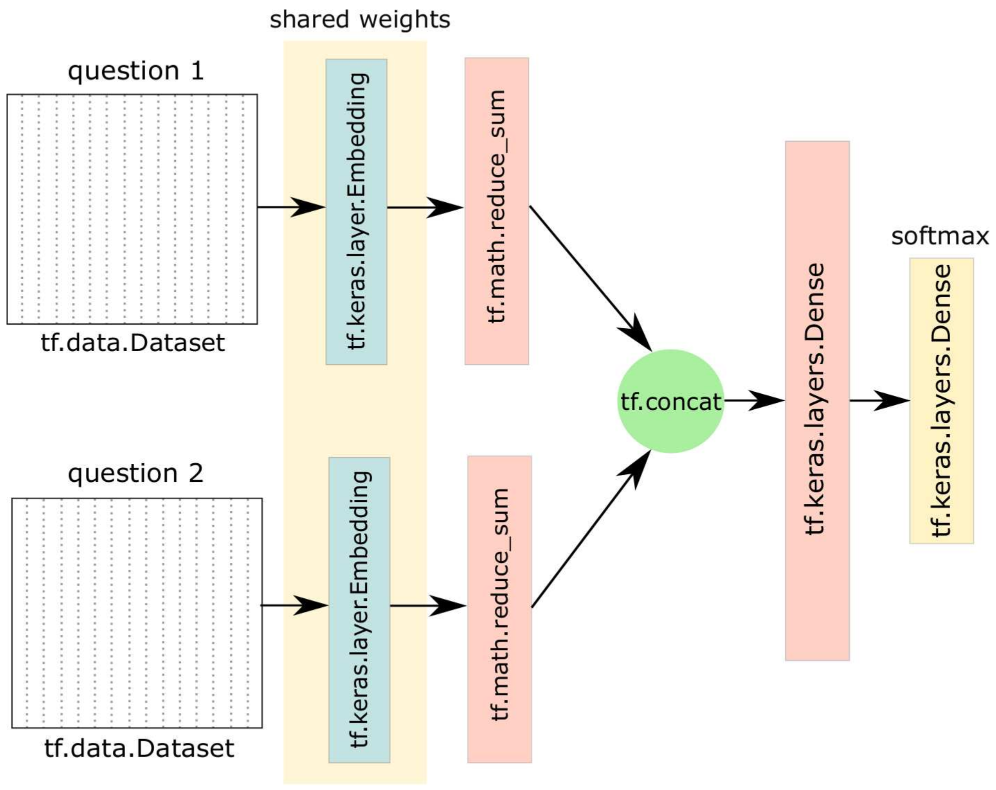


Figure 5.3: Neural network model based on a word embedding summation. Dropout layers are omitted in the picture for the sake of readability.

Variant	**Classes**	**BiLSTM**	**Dense**
BiDirLSTM1L3Cls	3	1	1
BiDirLSTM2L3Cls	3	2	1
BiDirLSTM1L2Cls	2	1	1
BiDirLSTM2L2Cls	2	2	1
BiDirLSTM2LDense2L2Cls	2	2	2
BiDirLSTM2LDense2L3Cls	3	2	2

Table 5.3: Variants of a BiLSTM encoder model.

using the pre-trained embeddings for text, whereas the code sequences are embedded using embeddings trained for code.

The sentence encoder is separated into two parts - an encoder for code and an encoder for text. The structure of both of them is identical since they are made up of two bidirectional LSTM layers. However, the encoders do not share their weights completely. The way how the weights are shared

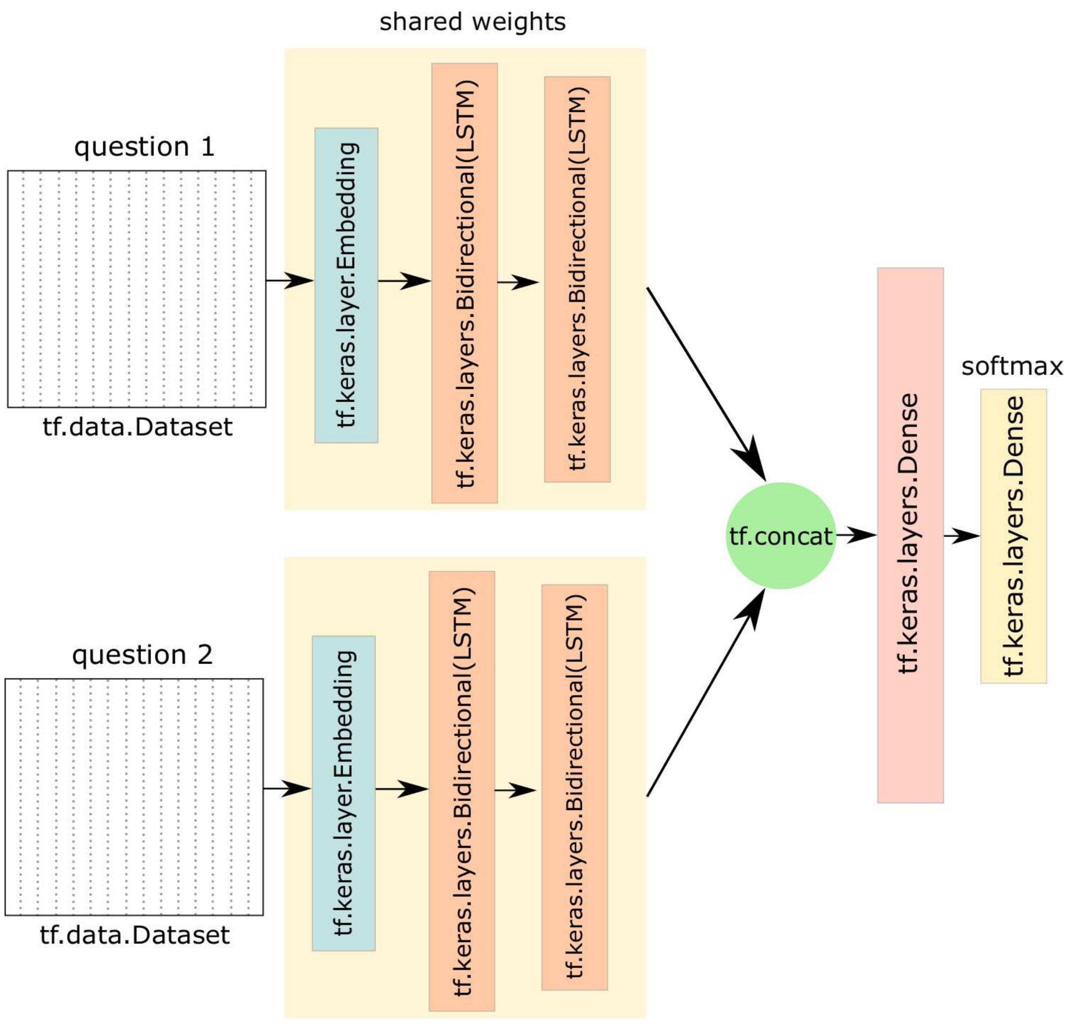


Figure 5.4: A bidirectional LSTM encoder model. Dropout layers are omitted in the picture for the sake of readability.

is marked out by yellow and blue background in the figure.

A result of passing the question through the embedding layer and the encoder is a vector representation of the code contained in the question as well as a representation of the text. These two representations are concatenated, and together they form a complete representation of the entire question. The fact that this model utilizes the complete information from the question is a significant advantage of this model. Thanks to that, the model can provide better results since it can distinguish, for example, the programming language of the code snippets.

To illustrate how the code processing can help to improve the accuracy of the model, imagine two questions where users ask "How to implement synchronization of threads t1 and t2 into the following code". For a model that only has the text of the questions, not the code snippet, these questions appear the same. However, if the code of the first query is written in Java and the second in C, the two questions are clearly different.

Finally, the representations of the questions are concatenated and processed by a softmax classifier with two dense layers and a softmax layer on the top. The code encoder model is trained in a variant with two and three classes (table 5.4).

Variant	**Classes**
BiDirCodeEncoder2L2Cls	2
BiDirCodeEncoder2L3Cls	3

Table 5.4: Variants of a BiLSTM encoder model.

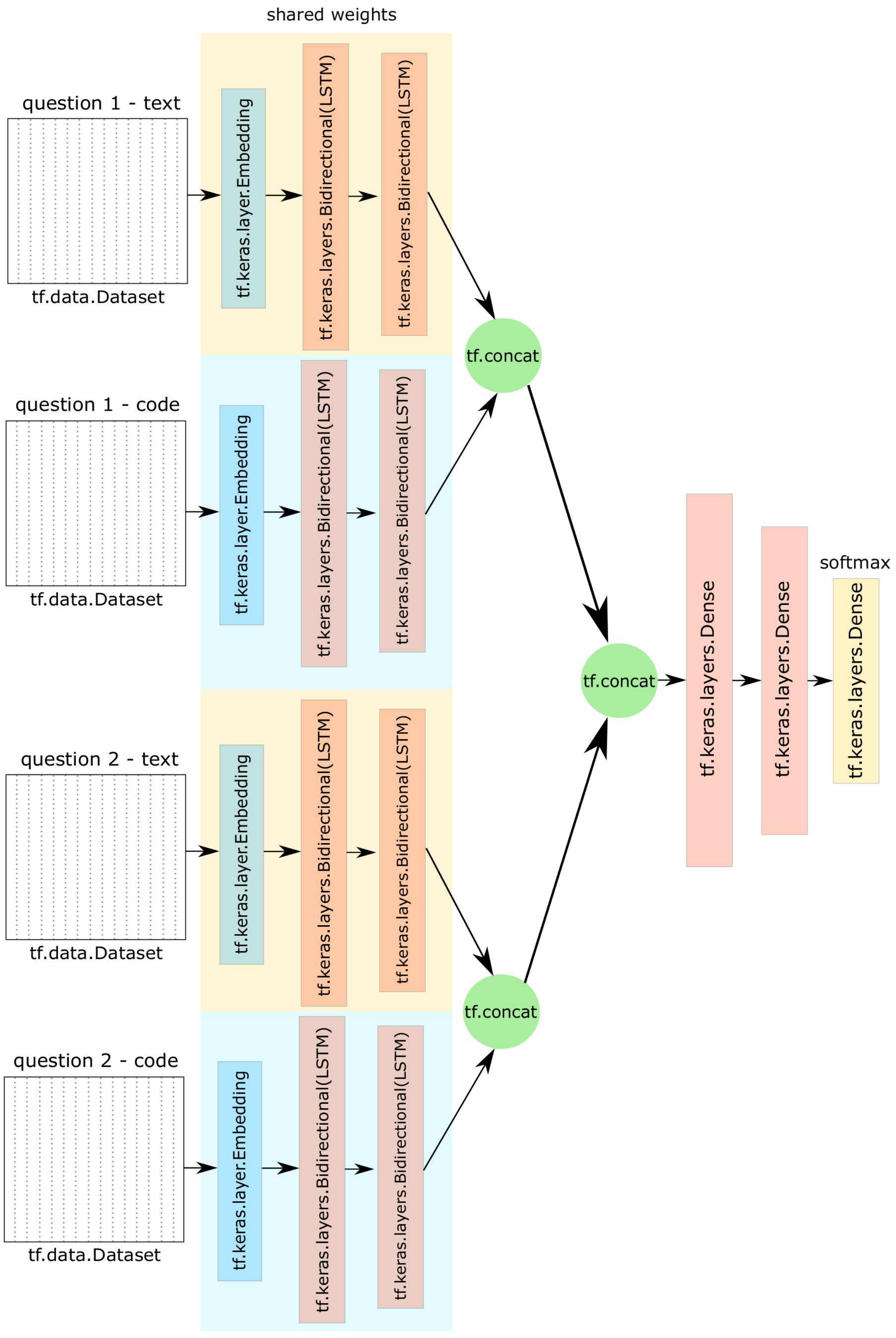


Figure 5.5: A bidirectional LSTM code encoder model. Dropout layers are omitted in the picture for the sake of readability.

6 Results and Evaluation

This chapter aims to discuss the results achieved using the models proposed in chapter 5. Furthermore, the chapter provides a more detailed description of realizing the chosen approaches. The first section discusses a data pipeline, while in the second section, the setup of hyperparameters is stated. Since numerous experiments are done for each of the model variants using different hyperparameters, only the setup leading to the best result of the given model is stated. After that, the second section states the achieved results and compares them with existing work on the SNLI (section 3.3.1) dataset. Finally, the results are further analyzed to provide deeper insights into the accuracy.

6.1 Data Pipeline Construction

This section describes the steps taken to prepare data for the neural network. In the first part, details of indexing the Stackoverflow data into Elasticsearch are discussed. In contrast, the second part describes how the data are exported from the Elasticsearch into a form usable by input pipelines.

6.1.1 Indexing Data into Elasticsearch

As stated in section 5.1, it is necessary to store the data in the Elasticsearch to assemble the dataset. Therefore, an Elasticsearch cluster made up of three nodes is established. The cluster is running on virtual machines provided by the MetaCentrum (`https://metavo.metacentrum.cz`). The machine running the cluster's master node also hosts a Kibana instance to facilitate exploration of the data in the cluster.

To index the data, a Logstash utility is used. Using the Logstash, one can configure an input pipeline that processes a given stream of data and ingest them into a configured Elasticsearch instance. The Logstash input pipelines used for this work consist of three primary operations - input, filter and output. The input operation ingests the given XML file into the pipeline. Then the filter operation takes care of parsing the XML and selecting desired fields. Finally, the output operation of the pipeline assembles a document and sends it to the Elasticsearch instance.

6.1.2 Dataset Export

According to a description in section 5.1, the dataset is assembled and resulting assignments of the posts are stored in the Elasticsearch database. The assignments are represented by dataset groups and post roles. In each group there is a *master post* and a number of *duplicate*, *similar* and *different* posts. Based on that, the dataset export is created.

The process of dataset export iterates over all the *master posts* and queries all remaining posts in the corresponding dataset group. Each of these posts is then used in a pair with the *master post* and together, they represent one example. The corresponding class is derived from the role of the post, not being the *master*.

The individual dataset examples are exported into a CSV file with format: "*first_post_ID*;*second_post_ID*;*class*". Furthermore, to be able to use a faster variant of the proposed input pipeline (section 5.3.1), two additional CSV files are created. One of them contains the corresponding preprocessed text of the posts and the second one consists of preprocessed code. The resulting dataset is split into three distinct parts, as shown in table 6.1.

Type	**Train**	**Dev**	**Test**	**Total**
Different	550 757	64 615	32 448	647 820
Similar	526 759	62 010	30 790	619 559
Duplicates	191 931	22 721	11 437	226 089
Total	1 269 447	149 346	74 675	1 493 468

Table 6.1: Stackoverflow dataset example count summary.

6.2 Experimental Setup

In the previous chapter, the architecture of the models is discussed. However, the setup of hyperparameters and other essential settings are not mentioned there at all. Therefore, this section discusses the hyperparameter setup using which the best results are achieved.

6.2.1 Embedding

As stated in chapter 5, the work utilizes pre-trained Word2Vec embeddings for both code and textual parts. These embeddings are joint for all the models and are trained on a whole set of posts from the Stackoverflow (chapter 4). Since experiments show that fine-tuning the embeddings during an end to end training does not improve the results at all, the pre-trained embeddings are fixed and are not fine-tuned during the training.

The embeddings are pre-trained using a CBOW variant of a Word2Vec (section 3.1.1) with negative sampling. The window size for training is set to five words. A dictionary for the textual embeddings is composed of all words that appear in a corpus at least 50 times. For the embeddings of the code parts, the minimum occurrence threshold is set to 500 occurrences.

6.2.2 Word Summation Model

The word summation model turns out to show the best results while using the two-class variant (*WordSum2Cls*) in combination with an f1 loss. Experimentally set hyperparameters are:

- batch size: 256
- length of input sequences: 150
- number of neurons in the first dense layer: 128
- the activation function in the first dense layer: ReLu
- L2 regularization factor in the dense and softmax layer: 0.05
- first dropout rate (after embedding): 0.5
- second dropout rate (all the others): 0.35

6.2.3 BiLSTM Encoder

The most successful BiLSTM encoder model is the two-class variant with two LSTM layers in the encoder and two dense layers preceding the softmax layer (*BiDirLSTM2LDense2L2Cls*). A used loss function is an f1 loss. Experimentally set hyperparameters are:

- batch size: 256

- length of input sequences: 150
- size of the first BiLSTM layer: 256
- size of the second BiLSTM layer: 128
- number of neurons in the first dense layer: 128
- number of neurons in the second dense layer: 64
- the activation function in the first two dense layers: ReLu
- L2 regularization factor in the dense and softmax layers: 0.05
- first dropout rate (after embedding): 0.5
- second dropout rate (all the others): 0.35

6.2.4 BiLSTM Code Encoder Model

The last model that uses a code encoder alongside the textual encoder shows the best results in the two-class variant with an f1 loss. The values of the hyperparameters are the same for both encoders and are therefore stated only once in a listing. The used hyperparameter values are:

- batch size: 256
- length of input sequences: 250
- size of the first BiLSTM layers: 256
- size of the second BiLSTM layers: 128
- number of neurons in the first dense layer: 256
- number of neurons in the second dense layer: 128
- the activation function in the first two dense layers: ReLu
- L2 regularization factor in the dense and softmax layers: 0.05
- first dropout rate (after embedding): 0.45
- second dropout rate (all the others): 0.3

Before presenting results achieved using these hyperparameters, it should be pointed out, that the dropout rates are configured to high values for all the models. Decreasing the rates results in significantly lower f1 scores. Additionally, it is important to disable a bias in the softmax output layer to prevent the models from learning a strong bias towards more numerous classes.

6.3 Model Results Evaluation

Tables 6.2 and 6.3 state achieved results of the proposed models on the Stackoverflow dataset. The tables are separate for two-class and three-class models since the results are not easily comparable, as discussed later.

Model	**Train**		**Dev**		**Test**	
	Acc	F1	Acc	F1	Acc	F1
BiDirCodeEncoder2L2Cls	88.0	75.8	86.9	74.6	86.1	74.1
BiDirLSTM2L2Cls	84.2	70.8	84.0	70.6	83.9	71.0
BiDirLSTM2LDense2L2Cls	84.0	70.8	85.5	70.7	84.6	70.9
BiDirLSTM1L2Cls	84.1	70.4	85.0	70.5	84.7	70.8
WordSum2Cls	84.3	70.1	84.5	69.2	86.2	68.8

Table 6.2: Results of two-class models.

Model	**Train**		**Dev**		**Test**	
	Acc	F1	Acc	F1	Acc	F1
BiDirCodeEncoder2L3Cls	88.2	81.6	86.6	80.2	85.3	79.2
BiDirLSTM2LDense2L3Cls	86.0	78.0	86.6	78.0	85.9	78.1
BiDirLSTM2L3Cls	85.6	77.6	87.4	76.8	87.2	77.5
BiDirLSTM1L3Cls	85.5	77.6	86.4	77.1	85.8	77.5
WordSum3Cls	84.6	76.8	85.1	77.0	86.7	75.1

Table 6.3: Results of three-class models.

The results show that the most successful models were the BiLSTM code encoders. That confirms the assumption that discarding the code contained in most queries can lead to a significant loss of information. Moreover, the word summation model was the least successful one, which was, however, expected since it is a baseline model. The difference in a test f1 score between the baseline models and BiLSTM code encoders is 5.3% and 4.1% for the two-class and three-class models respectively.

An interesting observation that can be done on the results is that the architecture variant of BiLSTM encoder does not play an important role in the resulting f1 score. In other words, it can be seen that neither an expansion of the softmax classifier part nor an expansion of the encoder improved the f1 score in a significant way.

Furthermore, it can be noticed that the three-class models report much better results than the two-class models. As a matter of fact, this observation might be misleading since a more in-depth analysis of confusion matrices shows that the two-class models work slightly better. That can be seen in figures 6.1 and 6.2. The figures show that despite the higher f1 score, the three-class models classify fewer duplicates correctly than the two-class models do. Even though this phenomenon is demonstrated on the BiLSTM code encoder, it applies to all the proposed models in the same way.

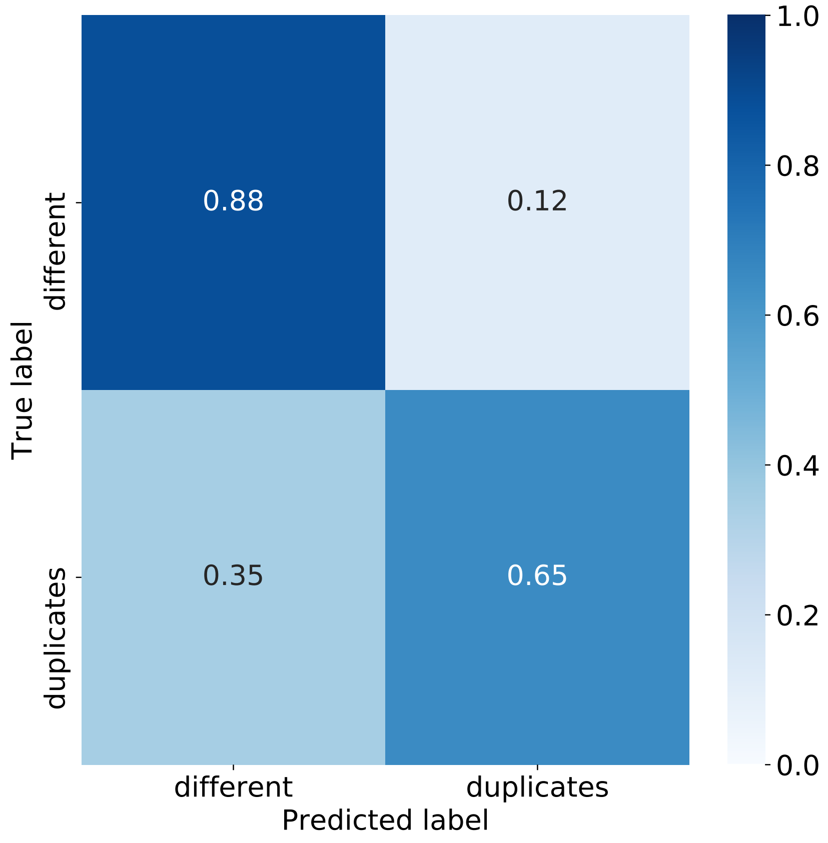


Figure 6.1: A confusion matrix for the two-class variant of the BiLSTM code encoder model on a test dataset split.

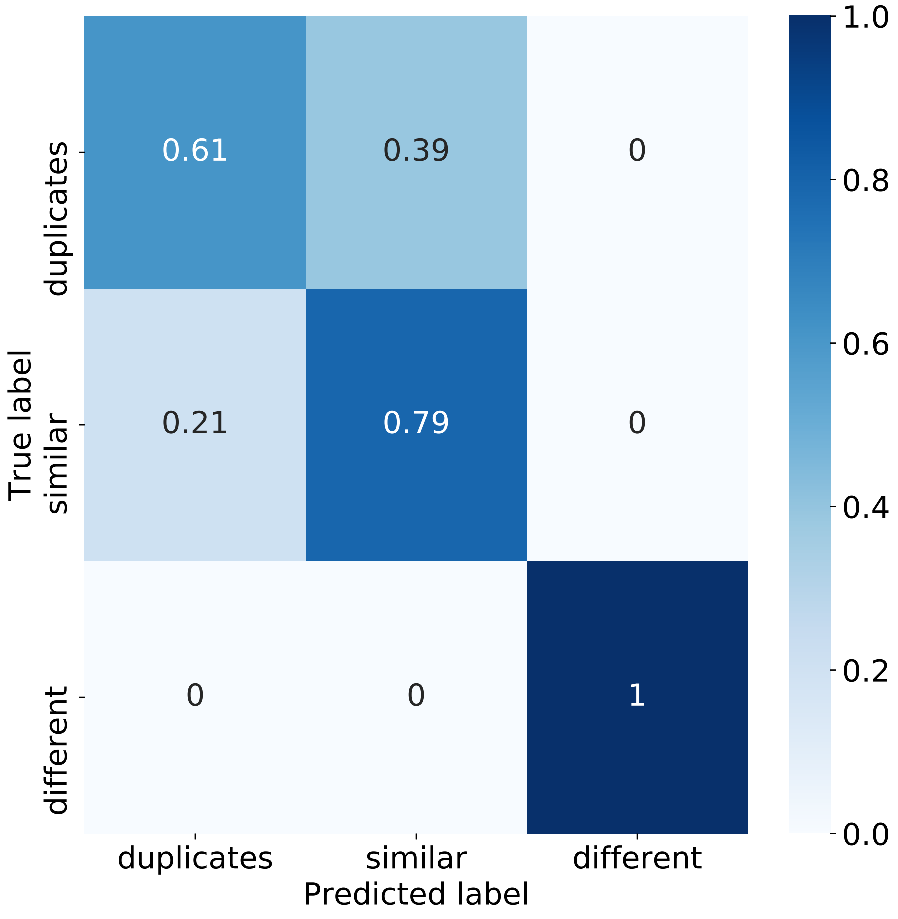


Figure 6.2: A confusion matrix for the three-class variant of the BiLSTM code encoder model on a test dataset split.

6.3.1 Other experiments

Developing and fine-tuning the model requires a lot of experiments and testing of various techniques. In this work, these are mainly a different loss function and a different merging step. This section briefly discussed the results and conclusions of these experiments.

An originally used loss function was a cross-entropy, which correlates very well with accuracy. Unfortunately, with the cross-entropy loss function, the models tend to learn a strong bias towards more numerous classes such as *similar* or *different*, not classifying the *duplicate* examples correctly. To address this problem, the f1 loss function is used instead and it brings a significant improvement observed in both the f1 score and confusion matrix. The improvement of the f1 score achieved by using the f1 loss is around 2%

Model	**Test accuracy**
LSTM RNN [34]	77.6
Sum of words [34]	75.3
BiDirLSTM2L3Cls	72.1
WordSum3Cls	69.7

Table 6.4: An accuracy comparison of the proposed models with the work [34] on the SNLI dataset.

on average.

Additionally, the work also investigated the possibility of using the absolute value of an element-wise subtraction as a way of merging the sentence representations. However, this merging technique turns out not to be working very well. As a matter of fact, the absolute difference worsens the f1 score by about 4%.

6.3.2 Comparison to SNLI

Classification on the SNLI dataset is, to some extent, similar to the task of recognizing duplicate questions. At least the proposed three-class architectures, except the BiLSTM code encoders, apply to this task, which is why the SNLI dataset has been used to verify the proposed models.

Since the SNLI dataset does not contain any code, the only models used for the verification on the SNLI are the word summation and BiLSTM encoder models. The results of these models (table 6.4) without any architectural change are significantly lower than those presented in [34] with similar models. However, this is expected since the models proposed in this work are tailored and fine-tuned to a slightly different task and the main aim of this work is to optimize the results for the Stackoverflow dataset, not for the SNLI.

6.4 Results Discussion

This section builds on the results presented in the previous section, discussing in more detail why the models can detect at most 65% of the duplicates. The discussion is mainly from the source data point of view since it can play a significant role in the achieved f1 score.

A first phenomenon that can be observed on the Stackoverflow website is that users often mark questions as duplicates, although they are not duplicates at all. Therefore the data source may contain a not negligible amount of false-positive examples, which can have a negative consequence on the network's metrics. On the other hand, the Stackoveflow policies require a question to receive three duplicate votes before actually being closed and marked as the duplicate. This rule dramatically reduces the number of wrongly labeled posts.

Additionally, as explained in chapter 4, the dataset contains semi-positive examples that are collected by Elasticsearch's full-text queries. During an assembly process, the query may find a duplicate that is not marked and is then erroneously treated as a non-duplicate example.

Last but not least, it can be expected that the Stackoverflow, despite an active community, contains a vast amount of unmarked duplicates. The Stackoverflow does not perform any automatic duplicate detection, which is entirely dependent on readers.

All the stated cases significantly affect the cleanness of the data and also reduces models' ability to learn to classify the question pairs correctly. Moreover, it is impossible to automatically filter out these wrong examples from such a huge data source, and therefore these facts shall be taken into considerations while reading the results.

7 Conclusion

In the scope of this work, a new dataset for semantic textual similarity was created from a Stackoverflow data dump. The dataset consists of almost 1.5 million examples and a learning target of the dataset is to classify whether two questions are duplicates or not. In the future, we plan to release the dataset as an official TF Dataset available for everyone. Furthermore, the work proposed three different siamese neural network architectures to solve this task.

The most successful architecture reports a test f1 score of 74.1%, improving the f1 score by 5.1% compared to a baseline model. An important feature of this architecture is an encoder of code in parallel to a textual encoder. This architectural element enables the network to utilize all the information stored in the data and to make more precise decisions. The proposed models in their hidden layers create a vector representation of sentences that can be obtained and used as a pre-trained sentence embedding for other tasks.

This work can be further followed by research into the usage of the obtained sentence representations by integrating them in a Stackoverflow information retrieval or code generation task. For future work, a web application for browsing and searching in the Stackoverflow data was constructed. At the moment, the application provides a standard full-text search only, but it is prepared for integrating the vector representation based search.

Furthermore, it might be beneficial to explore the usage of self-attention models such as BERT on the created dataset. Even though it would be very difficult to train BERT on the code tokens from scratch, it can be used for encoding the textual part alongside the current code encoder. Additionally, the Stackoverflow data contains questions with accepted answers, which could be utilized by another dataset with a learning objective to classify whether a post is an answer to a given question.

All scripts and neural network models that were used to create the dataset and train the models are available in public GitHub repository at the following URL:
`https://github.com/janpasek97/stackoverflow-siamese-network`.

List of Abbreviations

API - application programming interface

CBOW - continuous bag of words

CNN - convolutional neural network

DSA - Dynamic Self-Attention

FN - false negative

FP - false positive

GRU - gated recurrent unit

LSTM - long short term memory

ML - machine learning

MSE - Mean squared error

NLI - natural language inference

NLP - natural language processing

OOV - out of vocabulary

ReLU - Rectified linear unit

RNN - recurrent neural network

SNLI - Stanford Natural Language Inference

TF - Tensorflow

tf-idf - term frequency - inverse document frequency

TN - true negative

TP - true positive

Bibliography

[1] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning.* Massachusetts, MA, USA: The MIT Press, 2012.

[2] C. Nicholson. A beginner's guide to neural networks and deep learning. [Online]. Available: https://pathmind.com/wiki/neural-network

[3] F. Chollet, *Deep Learning with Python.* New York, NY, USA: Manning Publications Co., 2018.

[4] M. Nielsen, *Neural Networks and Deep Learning.* Determination Press, 2015.

[5] S. Raschka and V. Mirjalili, *Python Machine Learning.* Birmingham, UK: Packt Publishing, 2017.

[6] B. Karlik and A. V. Olgac, "Performance analysis of various activation functions in generalized mlp architectures of neural networks," *International Journal of Artificial Intelligence And Expert Systems*, vol. 1, no. 4, 2015. [Online]. Available: https://www.cscjournals.org/manuscript/Journals/IJAE/Volume1/Issue4/IJAE-26.pdf

[7] K. P. Murphy, *Machine learning: a probabilistic perspective.* Massachusetts, MA, USA: The MIT Press, 2012.

[8] (2017) Loss functions. [Online]. Available: https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html

[9] M. Haltuf. (2018) Best loss function for f1-score metric. [Online]. Available: https://www.kaggle.com/rejpalcz/best-loss-function-for-f1-score-metric

[10] M.-A. Maizas. (2019) The unknown benefits of using a soft-f1 loss in classification systems. [Online]. Available: https://towardsdatascience.com/the-unknown-benefits-of-using-a-soft-f1-loss-in-classification-systems-753902c0105d

[11] L. Bottou, "Stochastic gradient learning in neural networks," *Proceedings of Neuro-Nîmes 91*, 1991. [Online]. Available: http://leon.bottou.org/papers/bottou-91c

[12] G. Hinton. Overview of mini-batch gradient descent. [Online]. Available: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[13] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *The Journal of Machine Learning Research*, vol. 12, no. null, p. 2121–2159, Jul. 2011.

[14] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," *conference paper at ICLR 2015*, 2015. [Online]. Available: https://arxiv.org/pdf/1412.6980.pdf

[15] (2019) Multi-class metrics made simple, part ii: the f1-score. [Online]. Available: https://towardsdatascience.com/multi-class-metrics-made-simple-part-ii-the-f1-score-ebe8b2c2ca1

[16] Z. C. Lipton, "A critical review of recurrent neural networks for sequence learning," *ArXiv*, vol. abs/1506.00019, 2015.

[17] C. Olah. (2015) Understanding lstm networks. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs

[18] C. Olah and S. Carter. (2016) Attention and augmented recurrent neural networks. [Online]. Available: https://distill.pub/2016/augmented-rnns/

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3065386

[20] L. V. Utkin, M. S. Kovalev, and E. Kasimov, "An explanation method for siamese neural networks," *ArXiv*, vol. abs/1911.07702, 2019.

[21] A. Padmanabhan. (2019) Natural language processing. [Online]. Available: https://devopedia.org/natural-language-processing

[22] J. Eisenstein, *Natural Language Processing (draft version)*, 2018.

[23] J. Devlin and M.-W. Chang. Bert. [Online]. Available: https://github.com/google-research/bert

[24] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.

[25] X. Rong, "word2vec parameter learning explained," *CoRR*, vol. abs/1411.2738, 2014. [Online]. Available: http://arxiv.org/abs/1411.2738

[26] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," *EMNLP*, vol. 14, pp. 1532–1543, 01 2014.

[27] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *CoRR*, vol. abs/1802.05365, 2018. [Online]. Available: http://arxiv.org/abs/1802.05365

[28] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[30] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, Stanford, CA, US, 2019.

[31] E. A. Corrêa Júnior, V. Q. Marinh, and dos Santos Leandro Borges, "Nilc-usp at semeval-2017 task 4: A multi-view ensemble for twitter sentiment analysis," *NILC-USP*, pp. 611–615, aug 2017. [Online]. Available: https://www.aclweb.org/anthology/S17-2100

[32] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[33] http://ixa2.si.ehu.es/stswiki/index.php/stsbenchmark. [Online]. Available: http://ixa2.si.ehu.es/stswiki/index.php/STSbenchmark

[34] S. R. Bowman, G. Angeli, C. Potts, and C. D. Manning, "A large annotated corpus for learning natural language inference," *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2015.

[35] Y. Nie and M. Bansal, "Shortcut-stacked sentence encoders for multi-domain inference," *CoRR*, vol. abs/1708.02312, 2017. [Online]. Available: http://arxiv.org/abs/1708.02312

[36] A. Talman, A. Yli-Jyrä, and J. Tiedemann, "Natural language inference with hierarchical bilstm max pooling architecture," *CoRR*, vol. abs/1808.08762, 2018. [Online]. Available: http://arxiv.org/abs/1808.08762

[37] D. Yoon, D. Lee, and S. Lee, "Dynamic self-attention : Computing attention over words dynamically for sentence embedding," *CoRR*, vol. abs/1808.07383, 2018. [Online]. Available: http://arxiv.org/abs/1808.07383

[38] Z. Zhang, Y.-W. Wu, Z. Hai, Z. Li, S. Zhang, X. Zhou, and X. Zhou, "Semantics-aware bert for language understanding," *ArXiv*, vol. abs/1909.02209, 2019.

[39] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: A neural code search," *Proceedings of the 2nd ACM SIGPLAN*

International Workshop on Machine Learning and Programming Languages, p. 31–41, 2018. [Online]. Available: https://doi.org/10.1145/3211346.3211353

[40] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, p. 964–974, 2019. [Online]. Available: https://doi.org/10.1145/3338906.3340458

A Script Documentation

This appendix briefly describes implemented scripts and libraries that are used for the realization of the work. The scripts and utilities are organized into subchapters that correspond to a top-level repository structure. The scripts are accessible at:
`https://github.com/janpasek97/stackoverflow-siamese-network`.

A.1 Used Libraries

A complete list of libraries/packages and their versions necessary to run the scripts is in a file *requirements.txt* in the root of the repository. This section states only the most important ones only.

- **bs4** - stripping HTML tags from the post bodies
- **elasticsearch-dsl** - object-like access to Elasticsearch indices
- **html5lib** - stripping HTML tags from the post bodies (used by bs4)
- **matplotlib** - plotting graphs and confusion matrices
- **numpy** - vector computation (required by tensorflow)
- **pandas** - csv file operations and analysis
- **tensorflow** - neural network framework
- **tensorflow-datasets** - SNLI dataset source
- **tensorflow-hub** - pre-trained Word2Vec embeddings
- **gensim** - Word2Vec model training

A.2 Data

The *data* directory contains scripts and configuration files for indexing the data into the Elasticsearch instance and accessing them using the *elasticsearch-dsl* library.

index_config - *.conf

The directory *index_conf* contains five **.conf* Logstash pipeline configuration files. These pipelines are used for indexing the Stackoveflow dump into the Elasticsearch indices.

documents.py

The file *documents.py* contains class definitions for the *elasticsearch-dsl* library to be able to access the Elasticsearch documents as objects.

A.3 Dataset

The directory *dataset* contains scripts for assembling, exporting and cleaning the dataset.

dataset_cleanup.py

The script cleans invalid links from the Elasticsearch indices and removes an assignment of all documents to dataset groups.

make_ds.py

The script *make_ds.py* takes care of assembling the dataset. The script is separated into more parts since the process takes a long time. Therefore it shall be possible to restart the work from some point. A result of the script is a CSV file with post id pairs and labels. The procedure of creating the dataset is explained in chapter 4.

export_dataset_text.py

The script takes a CSV file with format "*first_post_id, second_post_id, label*" and outputs two CSV files. The first CSV has a format "*first_post_text, second_post_text, label*" and the second one has a format "*first_post_code, second_post_code, label*". The exported text and code is preprocessed and ready to be tokenized on spaces without any additional preprocessing.

shuffle_and_split.py

Provides functionality to shuffle the dataset and split it into three parts - train, dev, test. The module is used by *make_ds.py.*

A.4 Network

The directory *network* encapsulates all functionality that is necessary to create and train the neural network models.

assets

The directory *assets* contains the dataset exports, Word2Vec embedding matrices and word to dictionary index translation maps.

checkpoints

The directory *checkpoints* is expected to contain folders with checkpoints of the individual models. The subdirectories shall follow the naming convention "*modelname_loss*", since the script *evaluate_model.py* expects the model's checkpoint to be stored in such a directory.

logs

The directory *logs* contains all training logs for a Tensorboard.

losses/f1_loss.py

An f1 loss implementation as a child class of *tf.keras.losses.Loss*. The implementation is based on [9].

metrics

The directory *metrics* contains a custom implementation of a confusion matrix and f1 score, which is an enhanced version of the original Tensorflow code. Both implemented metrics are child classes of *tf.keras.metrics.Metric*.

models

The directory *models* contains definitions of the proposed models as a child class of *tf.keras.Model*.

utils

The directory *utils* contains many scripts with various functionality. These are, for example, configurations of the available models, text and code preprocessing scripts and dataset generators.

evaluate_model.py

The script *evaluate_model.py* creates the model selected by a command line parameter and loads its weights from the latest checkpoint. The created model is used for evaluating an accuracy, f1 score and confusion matrix on a test dataset.

main.py

The script *main.py* is used for training the models on the Stackoverflow dataset. It creates dataset generators, configures training callbacks and starts the training. The model to be trained, as well as the used loss function, is selected using command line parameters.

snli_baseline.py

The script *snli_baseline.py* is used for training the models on the SNLI dataset. It creates dataset generators, configures training callbacks and starts the training. The model to be trained, as well as the used loss function, is selected using command line parameters.

A.5 Word2Vec

The directory *word2vec* contains scripts for creating a text/code corpus and training the Word2Vec embeddings on the Stackoverflow data.

create_code_word2vec_ds.py

Creates a training corpus for training the Word2Vec embeddings of code tokens. The corpus consists of cleaned code snippets from all the Stackoverflow posts. An output of the script is a *.txt* file, where each line represents one training example.

create_text_word2vec_ds.py

Creates a training corpus for training the Word2Vec embeddings of textual tokens. The corpus consists of cleaned texts from all the Stackoverflow posts. An output of the script is a *.txt* file, where each line represents one training example.

create_dictionaries_and_embedding.py

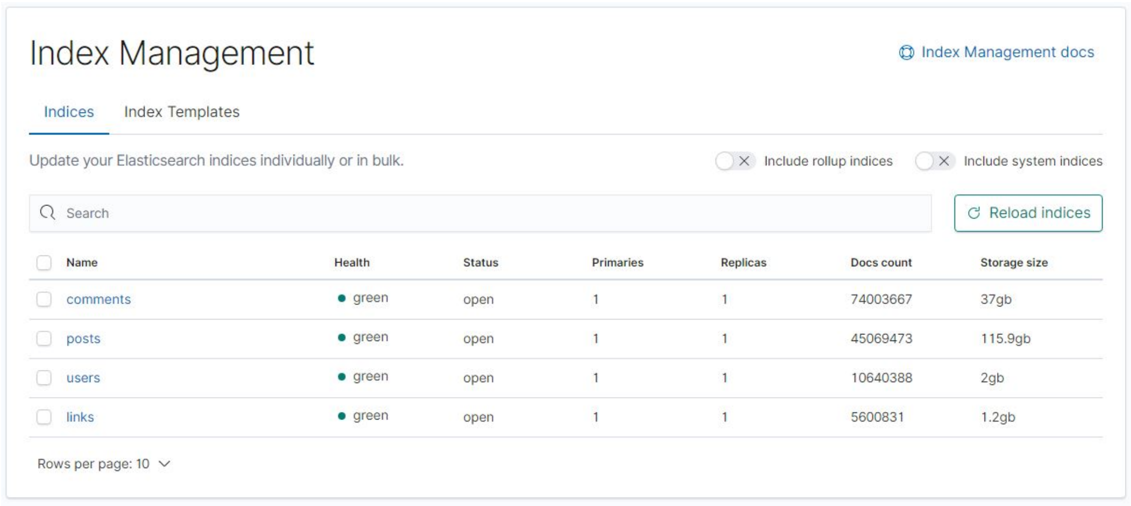
Exports an embedding dictionary and embedding matrix from an output of the Gensim Word2Vec model.

train_word2vec.py

Train Word2Vec embeddings on a given corpus using the Gensim library.

B Elasticsearch and Kibana Examples

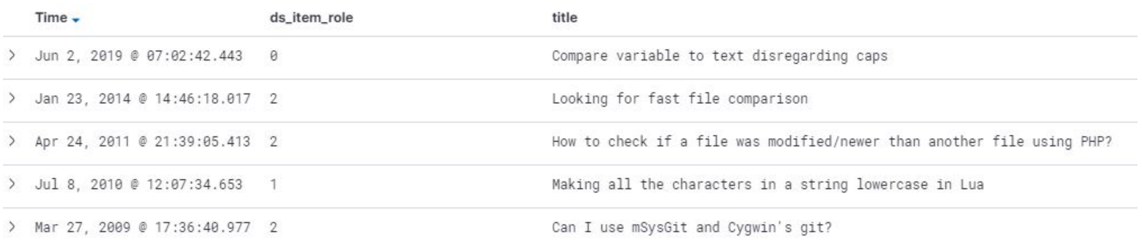
This appendix presents figures B.1 - B.4 that show screenshots obtained in a Kibana instance. The purpose of the screenshots is just to get an impression of how does exploring the data source indexed in Elasticsearch may look like. A reader is not expected to read the text in the figures.



Update your Elasticsearch indices individually or in bulk.

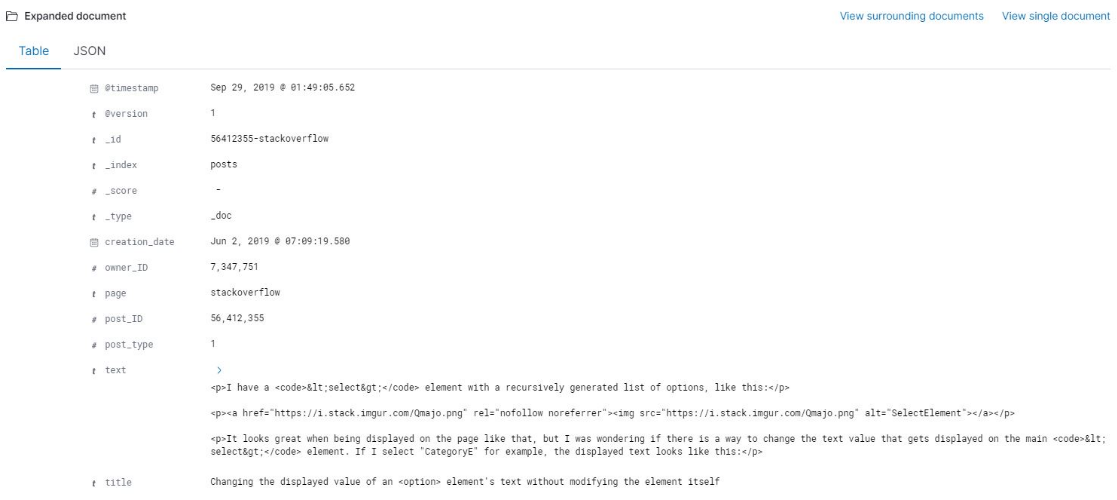
Name	Health	Status	Primaries	Replicas	Docs count	Storage size
comments	● green	open	1	1	74003667	37gb
posts	● green	open	1	1	45069473	115.9gb
users	● green	open	1	1	10640388	2gb
links	● green	open	1	1	5600831	1.2gb

Figure B.1: Summary of all documents indexed in the Elasticsearch cluster.



Time	ds_item_role	title
Jun 2, 2019 @ 07:02:42.443	0	Compare variable to text disregarding caps
Jan 23, 2014 @ 14:46:18.017	2	Looking for fast file comparison
Apr 24, 2011 @ 21:39:05.413	2	How to check if a file was modified/newer than another file using PHP?
Jul 8, 2010 @ 12:07:34.653	1	Making all the characters in a string lowercase in Lua
Mar 27, 2009 @ 17:36:40.977	2	Can I use mSysGit and Cygwin's git?

Figure B.2: One dataset group displayed in the Kibana instance. The figure shows a master post (*ds_item_role = 0*) with a corresponding duplicate (*ds_item_role = 1*) and three similar posts (*ds_item_role = 2*).



Expanded document

View surrounding documents View single document

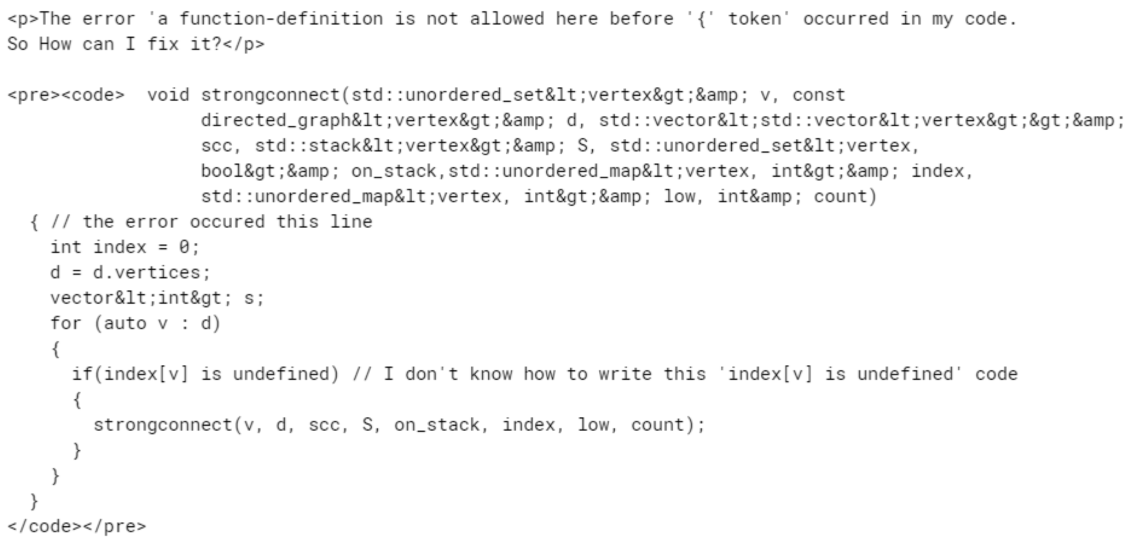
Table JSON

```
@timestamp      Sep 29, 2019 @ 01:49:05.652
@version        1
_id             56412355-stackoverflow
_index          posts
_score           -
_type           _doc
creation_date   Jun 2, 2019 @ 07:09:19.580
owner_ID        7,347,751
page            stackoverflow
post_ID         56,412,355
post_type       1
text            >
                <p>I have a <code>&lt;select&gt;</code> element with a recursively generated list of options, like this:</p>

                <p><a href="https://i.stack.imgur.com/Qmajo.png" rel="nofollow noreferrer"><img src="https://i.stack.imgur.com/Qmajo.png" alt="SelectElement"></a></p>

                <p>It looks great when being displayed on the page like that, but I was wondering if there is a way to change the text value that gets displayed on the main <code>&lt;
                select&gt;</code> element. If I select "CategoryE" for example, the displayed text looks like this:</p>
title           Changing the displayed value of an <option> element's text without modifying the element itself
```

Figure B.3: Expanded details of one document (post) displayed in the Kibana instance.



```
<p>The error 'a function-definition is not allowed here before '{' token' occurred in my code.
So How can I fix it?</p>

<pre><code>  void strongconnect(std::unordered_set&lt;vertex&gt;&amp; v, const
                 directed_graph&lt;vertex&gt;&amp; d, std::vector&lt;std::vector&lt;vertex&gt;&gt;&amp;
                 scc, std::stack&lt;vertex&gt;&amp; S, std::unordered_set&lt;vertex,
                 bool&gt;&amp; on_stack,std::unordered_map&lt;vertex, int&gt;&amp; index,
                 std::unordered_map&lt;vertex, int&gt;&amp; low, int&amp; count)
  { // the error occured this line
    int index = 0;
    d = d.vertices;
    vector&lt;int&gt; s;
    for (auto v : d)
    {
      if(index[v] is undefined) // I don't know how to write this 'index[v] is undefined' code
      {
        strongconnect(v, d, scc, S, on_stack, index, low, count);
      }
    }
  }
</code></pre>
```

Figure B.4: Example of an HTML body of one post.