Fachhochschule Graubünden
University of Applied Sciences

Churer Schriften zur Informationswissenschaft

Herausgegeben von
Wolfang Semar

Arbeitsbereich
Informationswissenschaft

Schrift 103

# Deep learning for detecting integrity risks in text documents

Urban Kalbermatter

Chur 2019

Churer Schriften zur Informationswissenschaft

Herausgegeben von Wolgang Semar

Schrift 103

# Deep learning for detecting integrity risks in text documents

Urban Kalbermatter

Diese Publikation entstand im Rahmen einer Thesis zum Master of Science FHO in Business Administration, Major Information and Data Management

Referent: Prof. Dr. habil. Albert Weichselbraun

Korreferent: Prof. Dr. Rolf Assfalg

Verlag: Arbeitsbereich Informationswissenschaft

ISSN: 1660-945X

Chur, September 2019

## Abstract

Deep Learning has become a widely used method in the field of Natural Language Processing including the field of text classification. It has been shown to perform better than conventional classification solutions in many cases. The focus of this thesis is to research and develop methods, which automatically identify discussions on integrity related issues in news articles using Deep Neural Networks.

A literature review is presented with a focus on the state of the art in Deep Learning for text classification. Further model architectures were identified, as well as frameworks to implement the models. The Deep Neural Networks were implemented, trained and evaluated. In an iterative process the networks were improved. Finally, based on the evaluation, recommendations for the implementation of Deep Learning methods for the detection of integrity risks were made.

# Index

## List of Figures

# 1. Intodruction

This thesis is part of the “Integrity Risk Monitor” project, which aims to advance integrity management by adapting methods from the fields of information and computer science. The project develops real-time integrity risk monitoring tools, with the goal

a. to support researchers in identifying and tracking integrity events,
b. to provide researchers with means to use data on past events, to develop models that are suitable to explain these events and to predict future gaps in integrity risk monitoring, and
c. to raise awareness of the affected stakeholders by publishing reports that summarize the results of these studies.

The focus of this thesis is on one particular task of the project: the research and design of methods, which automatically identify discussions on integrity-related issues. This is achieved by applying Deep Learning methods.

Since there is a multitude of different Deep Neural Network models, architectures and configurations, the aim is to identify, test and compare the most suitable ones for this particular task. Deep Learning is being used for a multitude of Natural Language Processing (NLP) tasks such as language modeling (Yoshua Bengio, Ducharme, Vincent, & Janvin, 2003), paraphrase detection (Socher, Lin, Ng, & Manning, 2011) and Word Embedding extraction (Mikolov, Chen, Corrado, & Dean, 2013) and has been shown to perform better than conventional classification solutions in many cases (Collobert et al., 2011). The performance of Deep Learning methods depends on both the quality and nature of the input data, the aim of the application, as well as the Deep Neural Network architectures and configurations being used. With the task being the detection of integrity events, this thesis aims to find out, which, among a selection of Deep Neural Network architectures and configurations performs best at classifying and thus detecting integrity risks in text documents.

# 2. Literature Review

## 2.1 Integrity

Integrity risks or integrity risk events are threats to the integrity of an organization or person, which could potentially lead to a decrease in public reputation – such as money laundering, corruption and fraud. This is usually referred to as “integrity violations” in the literature. The early detection of an integrity risk is a big advantage, because it gives the concerned parties the opportunity to react swiftly and accordingly to the threat, while it is still containable. (Molina, 2018, pp. 1–2)

## 2.2 Deep Learning

Deep Learning is a sub-category of Machine Learning, which is a sub-category of Artificial Intelligence. The field of Artificial Intelligence came to exist out of the question, if machines can be made to think like humans. Though this question is still unanswered, the field of Artificial Intelligence grew and is currently much broader than what it originally was, focusing on creating machines and software that are able to do intellectual tasks, which are normally done by humans. (Carbonell, Michalski, & Mitchell, 1983, pp. 69–79)

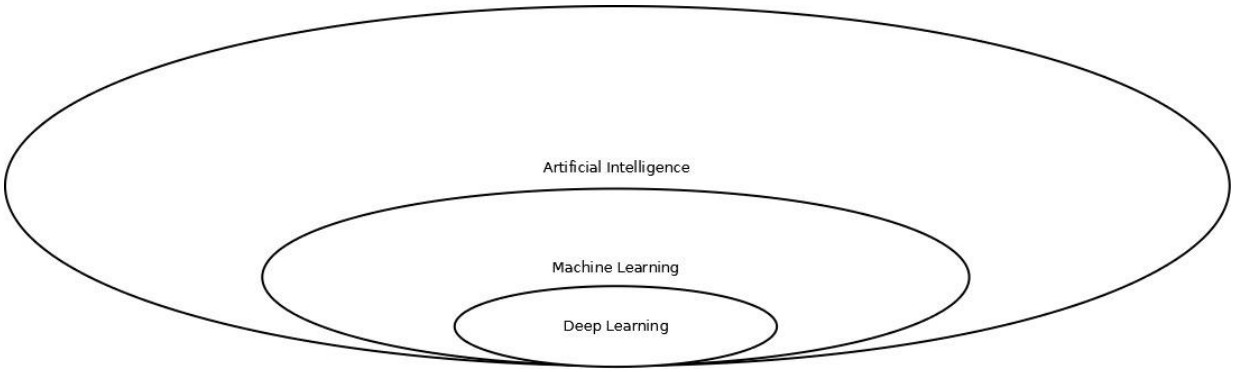



*Figure 1: Categorization of Deep Learning within the field of Artificial Intelligence – based on* (Chollet, 2017a, fig. 1.1)

These tasks can be achieved by non-Machine-Learning approaches like the symbolic AI approach, such as expert systems, in which a sufficient amount of explicit rules are defined to solve specific problems and tasks. These approaches are useful to solve well defined logical problems, but are not suitable to solve more complex and abstract tasks. The scope of these expert systems is the amount of pre-defined explicit rules. This makes expert systems more suitable for smaller tasks, because explicitly programming rules and managing big expert systems require a lot of time and effort. After the symbolic AI approach reached its peak success in the 1980s, it was clear that

an alternative approach was needed for more complex and abstract tasks. This made the Machine Learning approach gain in popularity and it soon replaced the symbolic AI approach as the most popular method in Artificial Intelligence. (Chollet, 2017a, Chapter 1; Schmidhuber, 2015, p. 4)

The difference between Machine Learning and classical programming, such as what has been used in the symbolic AI approach, is the following: The latter requires predefined rules and the output is a set of answers to a certain task, while in Machine Learning answers are provided and the algorithm finds rules that make the detection of these answers possible. (Chollet, 2017a, Chapter 1)

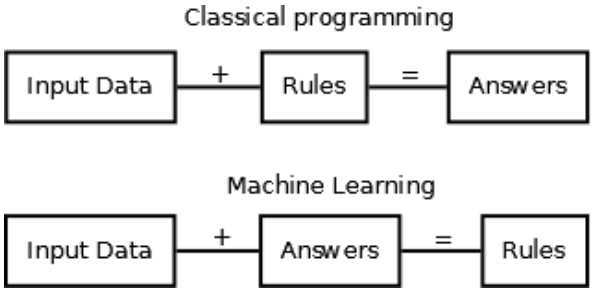



*Figure 2: the Machine Learning approach compared to classical programming - based on* (Chollet, 2017a, fig. 1.2)

In Machine Learning rules are learned rather than explicitly programmed. By using pre-labeled datasets consisting of input samples and answers to the specific task, the Machine Learning algorithm searches for statistical patterns, which can then be used to define rules. This process is called “training” and the output model containing these rules can be used as a classifier on unlabeled input samples or datasets to predict solutions to the same task it was trained for. (Chollet, 2017a, Chapter 1)

The main precondition to being able to classify a dataset with a Machine Learning approach is that the samples within the data are connected somehow by either correlation or causation to a certain label, such as a class. In addition to that, to be able to use Machine Learning to build an output model which is able to predict answers to the specific task, the following three requisites need to be met (Gluon Contributors, 2017, Chapter 1):

- Input data needs to be defined and available. Depending on the task this data can come in a variety of forms, in most cases it is either images, text, audio, video or structured data such as web pages. In supervised learning labels or

output examples need to be defined and linked to the input data. A single data point within a dataset is called a sample.

- A Machine Learning model which generates rules out of the input data has to be defined and build.
- A way to examine how successful the Machine Learning model is has to be defined. This makes it possible to compare the performance before and after an adjustment, through which can then be determined if the adjustments should be kept or not. This step is called learning and is done by using a loss function and an algorithm to minimize the loss function.

To make sure, that the training results are representative of data outside of the dataset, which is used for training, the labeled input data is usually split up into two parts, the training dataset and the verification or test dataset. The training dataset is used to train the output model, which subsequently will be validated by applying the output model to the validation dataset. The percentage of correctly solved tasks shows how well the output model performs on a dataset, which is different to the one used in training. It is thus important to keep track of two quantities: the training error and the test error or their counterparts, the training accuracy and the test accuracy. The training error is the error on the training dataset while the test error is the error on the verification dataset, meaning the amount of samples whose outputs were not correctly predicted, thus do not correspond to the output samples provided with the input data. The accuracy depicts the opposite, what percentage of samples in a dataset were correctly answered. (Gluon Contributors, 2017, Chapter 1)

Deep Learning describes a Machine Learning approach, which uses a multilayered Neural Network to solve a task. An Artificial Neural Network is a system, which is able to solve problems without explicitly programming it to do so based on a particular task. Artificial Neural Networks were first described in the literature long before Deep Learning, when in 1943 the concept was originally invented to understand and represent the information processing capabilities of a biological brain and thus of Biological Neural Networks. (McCulloch & Pitts, 1988)

An Artificial Neural Network consists of neurons, which are interconnected. A neuron is a single point in a Neural Network, which receives an input and computes an output by applying a mathematical function to it. Neural Networks used in Deep Learning are called Deep Neural Networks. Deep Neural Networks are a subset of Artificial Neural

Networks, which consist of a mix of layers stacked on top of each other. The difference between Artificial Neural Networks and Deep Neural Networks is that Artificial Neural Networks in their simplest form only have three layers, while the latter consists of multiple hidden layers.

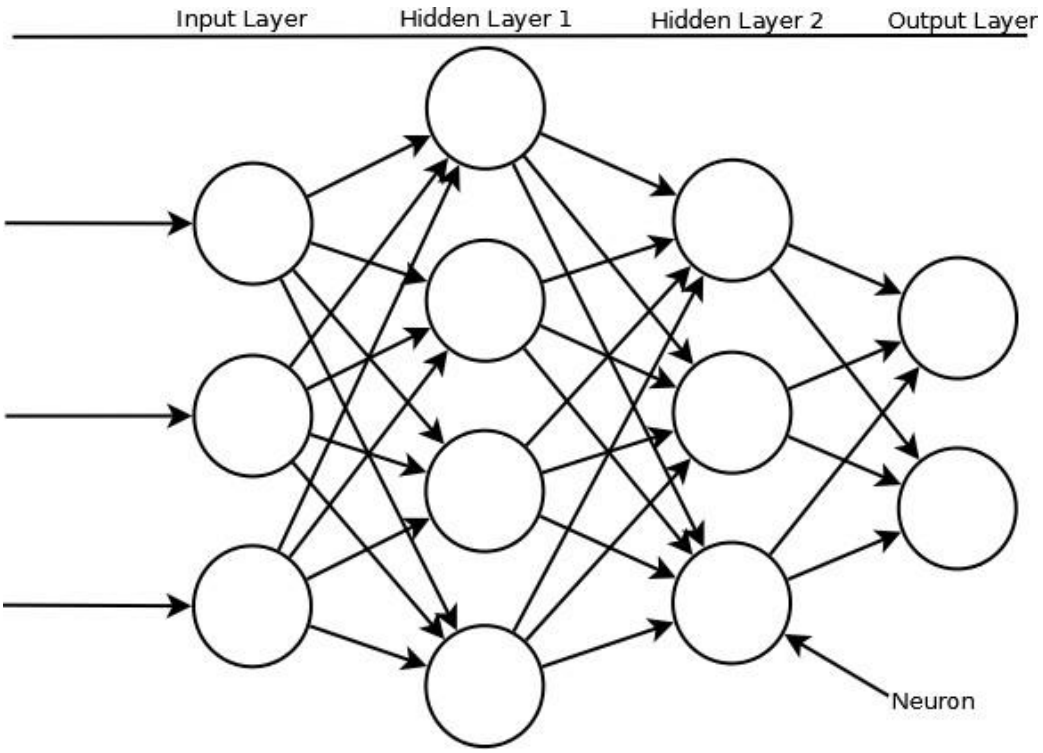



*Figure 3: Visualization of a simple Deep Neural Network architecture with two hidden layers – based on* (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014, fig. 1a)

The composition of a Deep Neural Network is as follows: The first layer is called the input layer, which takes in the input data. It is followed by one or multiple hidden layers, which are followed by an output layer. A layer takes in data, extracts features out of the data, which distinguish one input from another and sorts the inputs into different categories, depending if these features are present or not. By stacking multiple layers on top of each other, complex distinctions between different input data can be achieved. The amount of layers present in a Deep Neural Network is called the “depth” of the network. The more layers a Neural Network contains, the deeper it is, the less layers it contains, the shallower it is. This is also the reason why Deep Learning is called Deep Learning – It requires a deeper version of Neural Networks than the ones used in traditional Machine Learning. Many different architectures for these layers have been developed over the years to improve the performance of specific Deep Learning applications. (Chollet, 2017a, Chapter 2.1; Schmidhuber, 2015, p. 4)

The entirety of these layers is called a “model” or “network” (Chollet, 2017a, Chapter 1.1.4). In this thesis the word model is being used for the theoretical composition of a specific network, while a network is an actual manifestation of a certain model. This is not to be confused with the model, which is produced in the process of training, which is referred to in this thesis as the “output model”. The neurons in a network are either activated by the input data or the data forwarded by neurons in the prior network layer. By defining so-called weights or parameters to the individual layers and their neurons, the network calculates how to solve a certain task. At the beginning of the training, the layers will usually get randomly assigned weights. The goal of the training is to adjust the weights accordingly, so that the performance of the network improves. How much these weights get adjusted is defined by the following three parameters: (Gluon Contributors, 2017, Chapter 1)

- **the loss function**, also called cost function or objective, which measures success, thus allows to see if what was done was a success or not,
- **the gradient descent**, which is an optimization algorithm, which is responsible for calculating the extent of the changes to the weights on the basis of the results of the loss function,
- **and the learning rate**, which is a parameter that defines at which rate old information is overwritten by new information in each iteration.

The trainings consist of three repeating steps. First a prediction is made. For example, if sample A has feature B and C it means that sample A has label D. The predicted output score is compared with the actual output score, by using the loss function, which enables the network to evaluate its adjustments. If the score improved the weights get adjusted accordingly, if the score decreases the weights will also be adjusted accordingly to reflect this discovery. For example, sample A has label D as a predefined label, so the prediction was correct, this means the function is adjusted to reflect that the likelihood of an input sample having label D is higher, if it contains both feature B and C. This adjustment is done by using the gradient descent algorithm, which adjusts the function until a local minimum is achieved. These adjustment steps are called learning rate. The higher the learning rate, the faster the local minimum is achieved, but this also brings the risk of overshooting the local minimum. The mechanism used to make these adjustments possible is called the Backpropagation

algorithm. The total of these adjustments make up the output model. (Gluon Contributors, 2017, Chapter 1)

An additional parameter, which can be adjusted according to the specific needs at a time, is the amount of epochs during a training session. An epoch is one iteration through all the training data. The progress in each epoch is evaluated at the end of each epoch by applying the output model to the validation dataset. The trick is to let the Deep Neural Network go through as many epochs as needed but not more than that. If the training does not go through enough epochs, then the risk is, that the model is not adapted enough to the specific use case, but still resembles too much the initial random values. This problem is also called Underfitting and can be solved by continued training of the model. Too many epochs on the other hand lead to a problem called “Overfitting”. Overfitting occurs when a model is too adapted to the training set, thus loses the ability to generalize. (Hawkins, 2004).

The following factors make networks more susceptible to Overfitting: (Gluon Contributors, 2017, Chapter 2)

- **The number of degrees of freedom:** This is the number of tunable parameters in a Neural Network. If this number is higher, the network is more susceptible to Overfitting.
- **The value range of the weights:** If the range the weights can take on is wider, networks become more susceptible to Overfitting.
- **The size of the training dataset:** If the training dataset is small, the network is more susceptible to Overfitting, because the bigger the training dataset, the harder it is for the network to be over adjusted towards a big number of samples.

A way to measure the performance of a network and its output model and thus see if Overfitting is a problem or not, is to look either at the accuracy or error measure of each epoch and training. These values show the percentage of the trained and evaluated input datasets, which have been correctly predicted. For each epoch, there is an accurate measure for the training dataset as well as for the validation dataset. If the accuracy of the validation dataset does not change significantly anymore, but the one of the training dataset still does or if the accuracy of the training dataset grows significantly faster than the accuracy of the test dataset, the following epochs will likely lead to an output model which experiences Overfitting. (Chollet, 2017a, Chapter 4.4)

There is a family of techniques and measures to counteract Overfitting, which is called Regularization. Regularization consist of the following techniques and measures: (Gluon Contributors, 2017, Chapter 2)

- Making the model less complex by decreasing the amount of parameters. For example by leaving out certain input features that we know are not or less relevant for the task.
- Forcing the model to keep the weights small or limit the speed of their growth, for example by using a different loss function.
- Reinitializing the parameters if needed. Because the weights are randomly assigned, a network which experiences Overfitting could theoretically be the product of an unlucky set of initial weights.
- Using a special layer type called dropout layer, which disables certain neurons, thus allows the network to focus on other features.

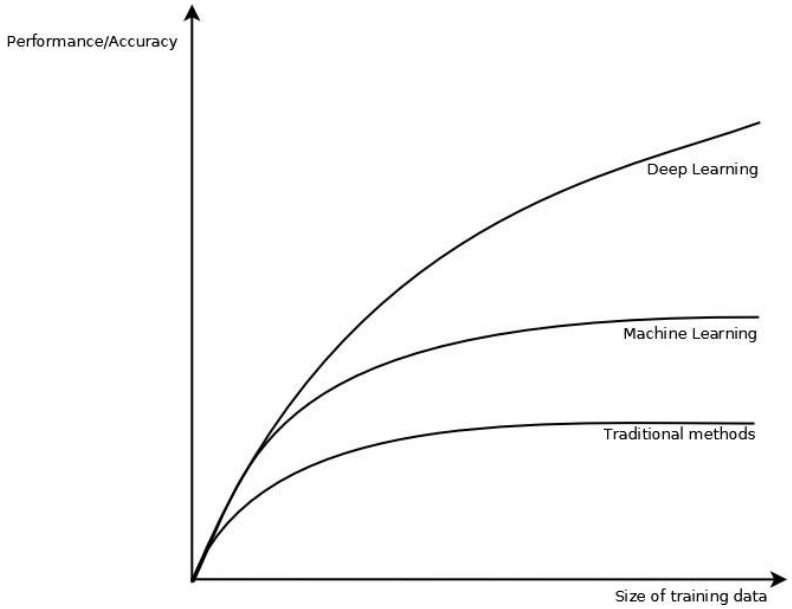



*Figure 4: Comparison of performance gain with an increase of training data size - based on (Ng, 2018, p. 12)*

Current areas of applications for Deep Learning include: (Gluon Contributors, 2017, Chapter 1; Pouyanfar et al., 2018)

- Face recognition
- Object recognition
- Gesture recognition
- Natural language processing
- Information retrieval
- Speech recognition
- Autonomous driving
- Machine Translation
- Ad Targeting
- Handwriting Transcription

- Data analysis

Deep Learning has been shown to perform equal or better in many of the above-mentioned areas of application (Collobert et al., 2011; Schmidhuber, 2015, pp. 18–25). Opposed to traditional methods the performance of Deep Learning depends heavily on the size and quality of the input dataset. The performance of Deep Learning increases more with a growing input dataset, than it does with traditional methods. With smaller input datasets the difference in performance are generally not significantly better (Ng, 2018, p. 12).

### 2.2.1 Deep Learning Modes

There are three different modes of Deep Learning: supervised, semi-supervised and unsupervised Deep Learning. Supervised Deep Learning consists of a known input and output, thus tries to find patterns in the input data of a certain output category, which isolates it from data of a different output category. Apart from categorization supervised learning is also used for regression and ranking of data. In unsupervised Deep Learning on the other hand, only the input data is known, without any classification or labels and the result is a classification based on categories found by the learning algorithm. This process is called clustering. Semi-supervised Deep Learning is a hybrid of both supervised and unsupervised Deep Learning. Usually in semi-supervised Deep Learning, there is a small set of input data with corresponding output data, as well as a larger set of uncategorized input data. Because of the high requirements on computing power, unsupervised learning was long neglected and only saw an increase in popularity in the last decade, with increasing and affordable computing power and advancements in Deep Learning research. (Schmidhuber, 2015)

Supervised learning includes the following tasks: (Gluon Contributors, 2017, Chapter 1)

- **Classification:** In classification we look for feature vectors, which can be certain shapes or colors in a picture or the grammar in a sentence. With the help of these feature vectors the data can be split into two or more classes. If there are only two classes, it is called a binary classification. A classification with more than two classes is called a multiclass classification. Examples for classification tasks are detection of cancers in CT images or a spam detection system for E-mail systems.

- **Regression:** In classification we ask the questions “what?” while in regression we ask “how many?” or “how much?” Here we try to predict a numerical output by applying statistical methods to a numerical input.
- **Tagging:** Tagging is a special form of multiclass classification, which allows one input to have multiple classes. An example for this is image or text tagging.
- **Search and ranking:** Supervised Machine Learning can be used to predict a ranking to a certain set of items. As an example the field of information retrieval deals with scoring, retrieving and ranking data.
- **Sequence learning:** In sequence learning we take a sequential input, such as a video, which consists of a number of sequential frames, where it is important to look at the context and thus the connection between each of them. Machine translation and speech recognition are also sequence learning tasks.

Examples for unsupervised Learning tasks are: (Chollet, 2017a, Chapter 4.1.2)

- **Dimensionality reduction:** Sometimes before a dataset is classified or analysed it needs to be cleaned and its size needs to be reduced. One way to do this is with Dimensionality Reduction, which reduces the amount of variables, by identifying the unimportant ones.
- **Clustering:** The aim of clustering is to group similar samples together. An example for Clustering is to find different customer segments in a customer database, based on not known features, to get a better understanding of similarities between different samples or groups of samples.

There are two other less prominent modes of Deep Learning called Reinforcement Learning and Self-supervised learning. In Reinforcement Learning the network will get inputs about its environment and tries to maximize some kind of reward by improving its actions. This is for example used in autonomous video gaming. Self-supervised Learning is similar to supervised learning, but it generates its own labels, usually by using heuristic methods on the input dataset. (Chollet, 2017a, Chapter 4.1)

### 2.2.2 Deep Learning Architectures

Among the most common Deep Neural Network architectures the two architectures Convolutional Neural Networks and Recurrent Neural Networks have been shown to be most successful for supervised NLP Tasks, thus they and some variations and a combination of them will be used and compared in this thesis. (Pouyanfar et al., 2018)

The following chapters contain a description of the functionality of these architectures as well as examples of how a model, which use these architectures, could look like and their fields of application.

*2.2.2.1 Convolutional Neural Network (CNN)*

A Convolutional Neural Network, also known as CNN or ConvNet, is a class of networks, which work with pattern recognition in data, which is represented in a grid-like form. This makes CNNs especially suitable for images, because pictures can be processed by representing each pixel as a value in a grid. In addition to CNN layers a CNN based network architecture typically consists of fully connected layers and pooling layers. Each CNN layer is typically followed by one pooling layer and the output layer is usually preceded by one or multiple dense layers. The reason for including pooling layers is to decrease the size of the network. Among the different types of pooling layer, max pooling layers have been shown to work best for CNN-based model architectures. (Lecun, Bottou, Bengio, & Haffner, 1998), (Chollet, 2017a)

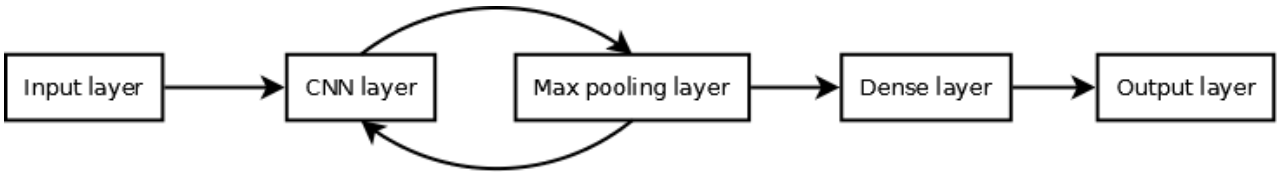



*Figure 5: A typical CNN architecture. In deeper CNN-based networks, the CNN layer and max pooling layer combination is repeated multiple times - based on* (Lecun et al., 1998)

A CNN is a dense Neural Network, which means that each neuron in one layer is connected to each neuron in the following layer. Fully connected layers are used to process data in different tensor shapes. Data in two dimensional tensor shapes is usually processed by some kind of fully connected network, such as a CNN. In Keras, which is the Deep Learning Framework used in this thesis, there are different types of CNN layers that can be used, depending on the amount of dimensions the shape of the data has. (Chollet, 2017a)

By extracting feature maps a CNN can perform a feature vector identification, which corresponds to the predefined output category. The disadvantage of CNN is their scalability, because CNN layers are fully connected layer, the needed computational resources to compute the trainings increases highly with deeper models and larger datasets (Fukushima, 1988; Lecun et al., 1998). The advantages of a CNN layer over a dense layer is, that features learned by a CNN layer are translation invariant, which

means that after features are initially identified, it can be recognized in any other location of other samples. The second advantage of CNN layers over dense layers, is that CNN layers can learn patterns through multiple layers. For example the first layer learns one characteristic of a certain feature, while the second layer learns more specific characteristics of this same feature. This allows CNN-based networks to detect more abstract features, when multiple CNN layers are stacked on top of each other (Chollet, 2017a).

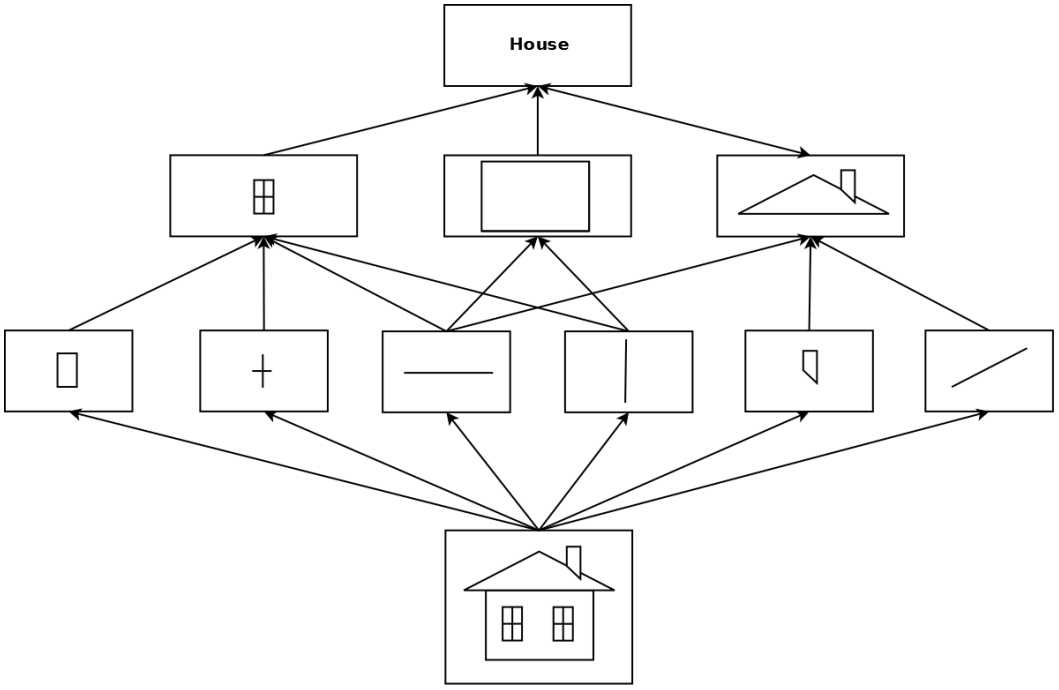



*Figure 6: A visualization of how CNN networks can learn patterns through multiple layers in image recognition – based on* (Chollet, 2017a, fig. 5.2)

The most well-known use case for CNN networks is in image and object recognition, because of its abilities to learn about single features through multiple layers. This is because visual data consists of many small shapes and different colors, which then can be connected to form a certain object. The AlexNet model, which is inspired by and build according to the model described above, achieved a historically high performance in 2012 in the ImageNet challenge, which is seen as the reference challenge in the field of image recognition. Another parameter, which made AlexNet more successful than its predecessors, was that it could overcome the Vanishing Gradient problem. Vanishing Gradient is a problem occurring in both CNN and RNN architectures, where the gradient gets so small, that it is preventing the weights from updating their values and thus stops the learning process of the network. AlexNet

overcame the Vanishing Gradient problem by using the ReLu activation function instead of Sigmoid functions (Krizhevsky, Sutskever, & Hinton, 2012). CNN has also shown to be efficient at solving NLP problems, primarily sentence classification and speech processing (Abdel-Hamid et al., 2014).

#### *2.2.2.2 Recurrent Neural Network (RNN)*

A Recurrent Neural Network, also called RNN or Feedback Neural Network are a subset of Recursive Neural Networks, as well called RNN or RvNN. RvNN can make predictions in tree-like structures. RNNs on the other hand work with sequential information, since they can store information of past sequences. This allows a RNN to see the full context of a single feature, instead of just the feature itself. Because of this mechanism, they are most commonly applied in text and speech processing and recognition. Since words change their meaning depending on their context, the networks short-term memory enables it to see and include this context. The context in this example would be the words before and after the words that are being processed, as well as its position in the sentence, instead of just the word itself.

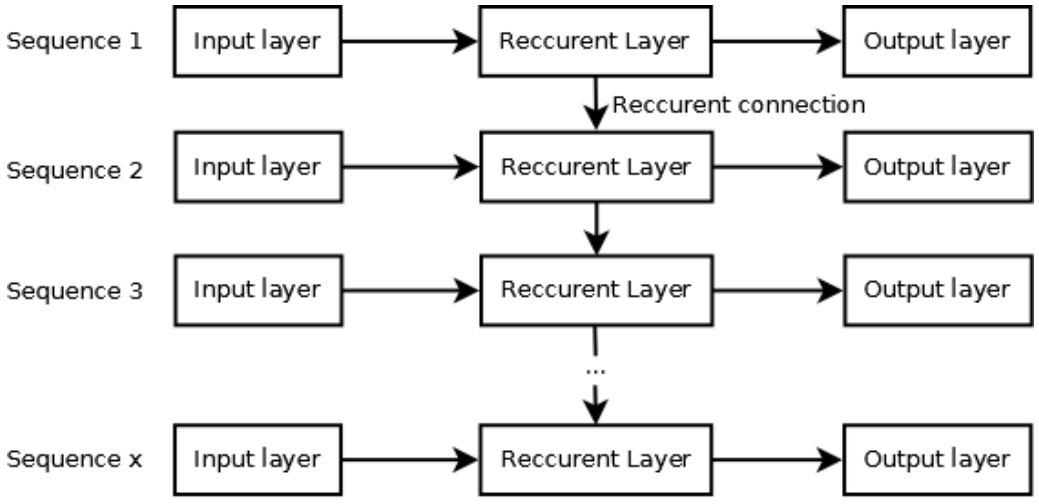



*Figure 7: A visualization of a simple RNN model. The recurrent connection allows the network to remember information of past sequences – based on* (Chollet, 2017a)

The disadvantage of RNNs is how difficult it is to store memories over long periods of time, because of their sensitivity to big chances during training in their short-term memory. This is due to a problem called Vanishing Gradient Descent or Vanishing Gradient problem, to which RNNs are susceptible to, due to the inheritance of the gradient because of their short-term memory function. If the gradient is small in a previous neuron, it influences the gradient of the following neuron, thus it will be even smaller. This leads to an exponential decrease or growth of the gradient, which makes

the network stop improving (Y. Bengio, Simard, & Frasconi, 1994). This issue halted progress for RNN-based networks until 1997 until it was solved by introducing memory blocks in the networks recurrent connections (Hochreiter & Schmidhuber, 1997; Lecun et al., 1998).

These memory blocks were named gates and replace the summation units in each of these layers. They are mechanisms, which decide which information to forget or add to the memory. By doing that the network can remember the important and forget the unimportant information and make predictions based on that data (Glorot & Bengio, 2010). The two most relevant architectures with this gate mechanism are called “Long Short-Term Memory” (LSTM) and “Gated Recurrent Unit” (GRU). The downside of adding these gate mechanisms is that they use significantly more processing power because of their complexity compared to a simple RNN (Cho et al., 2014; Li & Wu, 2015), (Pouyanfar et al., 2018).

The LSTM solves the vanishing gradient problem by using an input, forget and output gate in each sequence. The input gate defines how much of the new information should be kept, the forget gate defines does the same for the existing data in the memory and the output gate decides what of the current memory state should be shared with the following sequence. GRU solves the problem with just two gates, a reset gate and an update gate. The update gate is responsible for deciding to what degree the memory is being updated, while the reset gate is responsible for resetting the computed state by forgetting the previously computed state. (Chung, Gulcehre, Cho, & Bengio, 2014)

LSTM has shown to be efficient at solving tasks, which require looking at long-term dependencies (Graves, 2012, Chapter 4). Because of the similar structure and functionality as well as similar results for tasks based on long-term dependencies, it can be assumed, that this is also true for GRU. GRU, which is a recently developed architecture, seems to be a more efficient way of achieving similar results for these tasks than with LSTM. There is no clear evidence towards which of the two gate mechanisms is superior in their performance, but both of them clearly outperform a vanilla RNN, while GRU is generally faster (Chung et al., 2014).

#### *2.2.2.3 Recurrent Convolutional Neural Network (RCNN)*

Recurrent Convolutional Neural Networks are a combination of RNN and CNN. RCNN have been shown to perform better than RNN and CNN in both object recognition and

text classification (S. Lai, Xu, Liu, & Zhao, 2015; Ming Liang & Xiaolin Hu, 2015). RCNN can be implemented with both LSTM and GRU RNN architectures, as well as RNN layers without a gate mechanism. An alternative name for the LSTM RCNN is LRCN (Donahue et al., 2017).

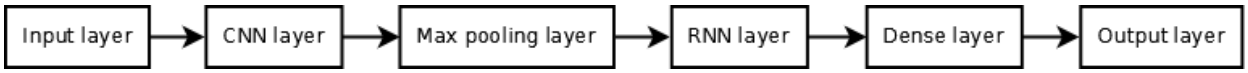



*Figure 8: A typical RCNN model based on* (Li & Wu, 2015)

The input layer in a RCNN model is usually followed by the CNN layer and a max pooling layer, as described in chapter 4.2.3.1, which is followed by the RNN layer, a dense layer, as well as the output layer. The output layer is generally either a dense layer or if needed a dropout layer. (Li & Wu, 2015)

#### *2.2.2.4 Bidirectional Recurrent Neural Network (BRNN)*

A special type of RNN are Bidirectional Recurrent Neural Networks also called BRNN, which not only store information from past sequences, but also from future sequences. This is done by training the network in both time directions simultaneously, by splitting the state of neurons in a RNN network, so that one part is responsible for the positive time direction and the other for the negative time direction. The outputs of both parts of the network are usually merged after each layer, although summation after multiple layers is also a possibility. (Schuster & Paliwal, 1997)

This structure provides the advantage of increased accuracy and an increase in detected contextual features (Chollet, 2017a, p. 207). Since BRNN violate causality, it cannot be used on temporal data, which cannot provide this, such as navigation tasks or financial predictions. For spatial data, such as the data used in this thesis, on the other hand, this is not a problem (Graves, 2012, Chapter 3.2.4).

The architectures used for the BRNN Deep Learning models are the same as for other RNN models, since apart from their bidirectional layout, their functionality and architecture is the same.

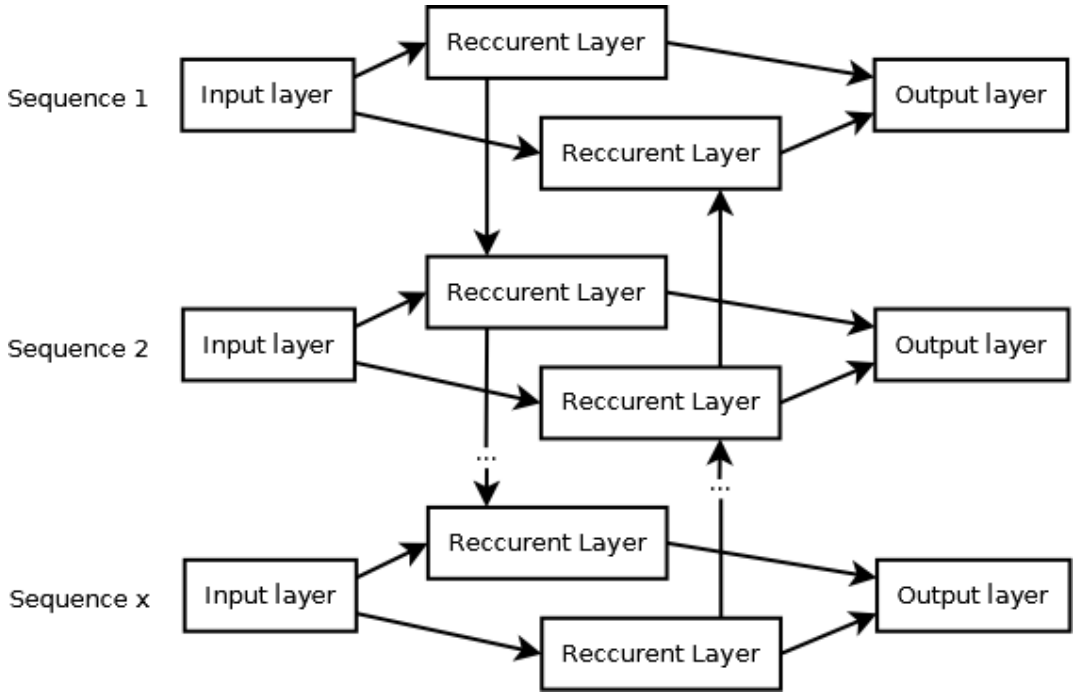



*Figure 9: Visualization of a Bidirectional RNN. The recurrent connections are shared in both the forward and backward direction – based on* (Schuster & Paliwal, 1997, fig. 1)

### 2.2.3 Neural Network Layers

A network or model consists of a sequence of layers. On top of the above-mentioned Deep Neural Network architectures, the following layer types are used to build and enhance the networks and their performance. The below-mentioned types are only a selection of many more layer types being used in a Deep Neural Network.

#### *2.2.3.1 Dense Layers*

A dense layer, also referred to as fully connected layer, is a regular and linear Neural Network layer, which is densely connected, which means that each neuron receives inputs from every other neuron in the previous layer and each of the inputs is connected to each of the outputs by a weight. This layer is usually followed by a non-linear activation layer function. (Chollet & others, 2015)

#### *2.2.3.2 Activation Layers*

Activation layers compute which neurons to fire by applying an activation function on the respective input (Chollet & others, 2015). An activation function calculates the weighted sum of its inputs and adds a bias.

The following is a selection of activation functions: (Glorot, Bordes, & Bengio, 2011; Nwankpa, Ijomah, Gachagan, & Marshall, 2018)

- **Step function:** The output of a step function, also called a binary function, is always either 0 or 1, depending on the value of x. A step function is not applicable to Deep Learning, because the Gradient Descent algorithm cannot update the weights due to their binary nature, thus cannot improve its performance.
- **Linear function:** Contrary to a step function, a linear function is not restricted in its range. Linear functions are rarely used because their linearity limits their usefulness for Deep Learning applications. However, their advantage is, that they offer high performance and are easy to optimize with Gradient Descent methods. The Rectified Linear Unit activation function, also called ReLU function, is a nearly linear function, thus benefits of these properties of linear functions without running into a Gradient Descent problem, by rectifying values below zero, by making them 0, thus making them unusable to the network.
- **Logistic function:** A logistic activation function has a range between 0 and 1. It has the shape of the letter S. The problem with logistic activation functions is, that they do not work well with Gradient Descent. This is because they tend to be zero-centered, which makes the network too sensitive to the Gradient Descent problem. An example of a logistic activation function is the Sigmoid function. Sigmoid functions are mostly used in feedforward Neural Networks to get a binary predicting probability based output. Another logistic function is the Softmax function, which compared to the Sigmoid function is mostly used for multiclass classification tasks.
- **Hyperbolic tangent function:** An example of a hyperbolic tangent activation function is the Tanh function. The Tanh function has a range of -1 to 1 and has been shown to perform better than the Sigmoid function. The problem with the Tanh function is, that they can only achieve a gradient of 1 if the input value is 0, which renders some neurons unusable by setting their weight to 0. This is a condition called dead neurons. The Tanh function is mostly used in language modelling and speech recognition.

The most commonly used activation functions used in Deep Learning are Sigmoid, ReLU and Softmax. The current trend is to use ReLU for the hidden layers to counteract the gradient descent problem and Softmax or Sigmoid for the output layer. Which one of the latter two is used for the output layer, depends on the desired output.

Sigmoid is used, if the output is a binary classifier and Softmax is used if the output is a multiclass output. (Nwankpa et al., 2018)

In Keras activation functions can either be used by adding an activation layer to the network or by adding the activation argument to any forwarded layer. Keras supports all of the most well-known activation functions and allows the import of custom activation functions. (Chollet & others, 2015)

*2.2.3.3 Dropout Layers*

The purpose of a dropout layer is to counteract Overfitting. Dropout layers have shown to be successful in counteracting Overfitting in both CNN and RNN based networks. A dropout layer does this by setting the output of a selection of single neurons to 0. This makes neighboring neurons more sensitive and increases the size of changes in the weights of these neurons. The dropout layer is inserted before a linear or nearly linear activation function. (Srivastava et al., 2014)

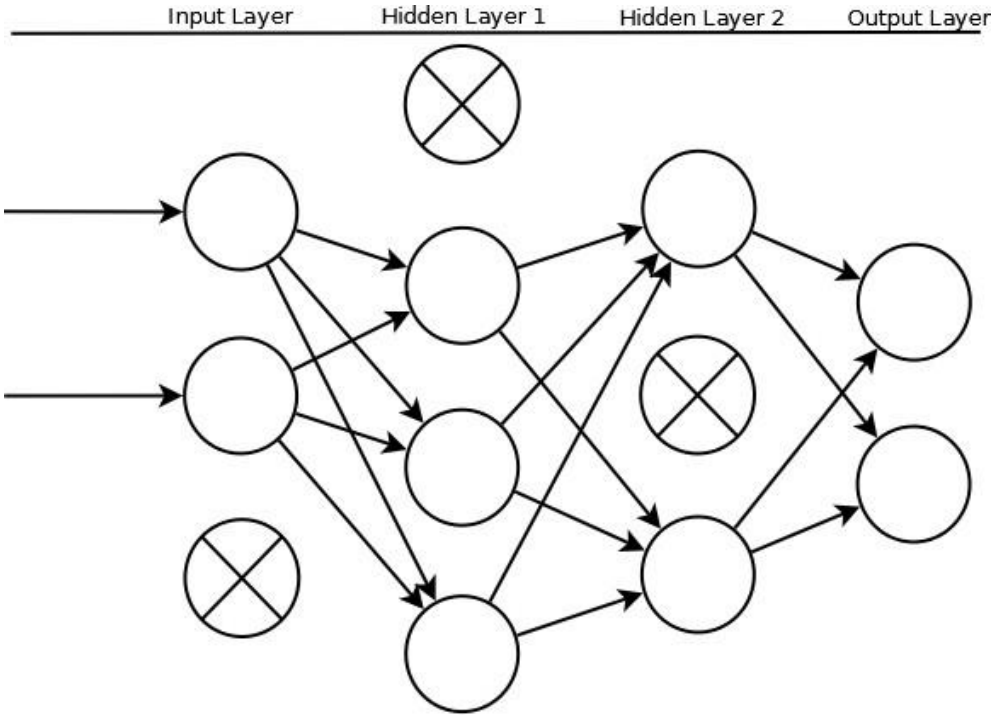



*Figure 10: Example of a Neural Network with two hidden layers after applying dropout – crossed neurons have been dropped, so that others gain more sensitivity – based on* (Srivastava et al., 2014, fig. 1b)

The original paper applies the dropout function on each dense layer before the output. The same can be applied to LSTM networks (Zaremba, Sutskever, & Vinyals, 2014). Recent research shows, that it is also possible to use dropout layers after the activation

layer of a CNN layer, although at a smaller rate of 0.2 instead of 0.5 (Park & Kwak, 2017). The dropout layers take a 1-dimensional tensor as an input, so a flatten layer has to be used beforehand to adjust the tensors if they are higher dimensioned.

*2.2.3.4 Pooling Layers/Global Max Pooling Layers*

Max pooling layers are used to aggressively downsample networks and thus to reduce the number of neurons to compute. Usually pooling layers reduce the amount of neurons by a factor of 2. This down sampling process increases the network's ability to generalize features. In Keras there are different pooling layer types, which have to be chosen depending on the dimensionality of the tensors being forwarded by the previous layer. Additionally pooling layers can either be global, average and/or max. Global pooling layers down samples the network to one single value, which is the most prominent feature. Average pooling layers downsample the network to the average value. Max pooling layer on the other hand downsample a network to its maximum values. (Chollet, 2017a)

In Keras the following parameters are available for global max pooling layers:

- **pool_size:** This defines the size of the max pooling windows.
- **strides:** This defines the factor, by which the networks should be downsampled by.
- **padding:** If this is set to "valid", it ensures both that the data does not shrink in its dimensionality and that input data on the border of the input grid (for example in a vectorized image representation) is not disadvantaged, because they get looked at less thoroughly, because their position is part of less filter regions.
- **data_format:** This defines in which order the dimensions of a tensor are being outputed. This is relevant if the input data is in the shape of a 3D tensor.

The output of a global max pooling layer is defined by the value given in the parameter data_format. (Chollet & others, 2015)

*2.2.3.5 Word Embedding Layers*

A Word Embedding is a vector representation of words and their semantic similarity (Mikolov et al., 2013). Word Embeddings have been shown to improve the performance of NLP tasks in Deep Learning (Socher et al., 2013). There are two ways in which an embedding can be implemented in a Deep Neural Network. The first one is to use an embedding layer, which makes the model use an unstructured random

vector that then will be adapted to the task. The embedding layer requires integer encoded input data, so that each word in the input data is assigned to a unique identifier. The quality of these embeddings depend strongly on the size of the training set. If not enough data is available to train a Word Embedding adapted to the task, the second possibility is to use a pre-trained Word Embedding. A Deep Learning model usually starts with one of the above-mentioned embedding layers (Chollet, 2017a).

The embedding layer in Keras can be used as a layer in a network or as a stand-alone module. If it is used as such, the Word Embedding can be saved for later use, without having to rebuild it with each new network. The embedding layer in Keras also allows to use pre-trained generalized Word Embeddings. The embedding layer also has weights on its own and the output is a 2D vector, where each word is represented by a unique integer. The embedding layer must specify the following three parameters:

- **input_dim:** This is the number of words contained in the embedding.
- **output_dim:** This is the size of the output vector.
- **input_length:** This is the size of the input data in words.

There are two popular methods of creating Word Embeddings from a text dataset. The first one is called Word2Vec, the second one GloVe. Both of them are unsupervised learning algorithms to extract vector representations of text data. (Pennington, Socher, & Manning, 2014), (Mikolov et al., 2013)

In Keras it is possible to use GloVe as well as Word2Vec Word Embeddings. To use Word2Vec the library Genism has to be used. While both algorithms compute the models in a different way, the output model is only slightly different and can be converted if the Deep Learning Frameworks does not support them. (Chollet & others, 2015)

### 2.2.4 Deep Learning Data Representations

There are different ways in which input data can be represented in Deep Neural Networks, mostly depending on the kind of input data provided. The data structure in Deep Learning is always a so called tensor, which is a container for numbers, that is used as a representation of the input data in a Deep Neural Network. A vector is usually defined by its number of axes, its shape and its data type. The following list shows the different types of tensors according to their dimensionality: (Chollet, 2017a, Chapter 2.2)

- **Scalars:** A scalar is a tensor that contains only one value, thus is zero-dimensional.
- **Vectors:** A vector is a tensor which contains values on one axis, thus is a one-dimensional tensor.
- **Matrices:** A matrice is a vector which contains two axes, thus is a two-dimensional vector.

Additional axes can be added to create multi-dimensional tensors. The shape of a tensor is defined by the size of each axis. The shape of a tensor is usually depicted by round parenthesis, with the amount of numbers being the dimensionality and the size of each axis being the numbers. For example the vector (3, 5) is two dimensional, because there are two numbers within the parenthesis, thus the vector is a matrice. The size of the two axes is 3 and 5 respectively. The data type is defined by the type of data contained in a tensor. The most commonly used data types in Python are float32, uint8 and float64. (Gluon Contributors, 2017, Chapter 1)

### 2.2.5 Deep Learning Frameworks

There is a multitude of different Deep Learning frameworks. The following chapters give an overview over a selection of frameworks. Deep Learning frameworks come in two main forms: low-level and high-level frameworks.

#### *2.2.5.1 Low-Level Deep Learning Frameworks*

Low-level Deep Learning Frameworks need a lot of coding experience and in-depth understanding of the underlying mechanisms to be applicable. On the other hand, this allows the developer to fine tune even small details, which makes them well suited for research and for the development of new Deep Learning models from scratch.

The following table shows an overview over the most popular low-level Deep Learning frameworks.

| **Framework** | **Written in** | **License** |
|---|---|---|
| Tensorflow | C++, Python | Apache License 2.0 |
| Torch | C, Lua | BSD License |
| Theano | Python | BSD License |
| MxNet | C++ | Apache License 2.0 |
| Deeplearning4J | Java | Apache License 2.0 |

Additional information about each framework can be found within the following chapters.

#### 2.2.5.1.1 Tensorflow

Tensorflow is a Machine Learning library which is being developed by the Google Brain team and has been published under an Apache License 2.0. Tensorflow is being used as the main AI library for many Google Services, such as Google Street View, Google Translate, Gmail, Google Photos and Google Search and is being used as the backend for many High-level Deep Learning frameworks. Tensorflow is written in Python, but uses both Python and C++ in its implementation. Interfaces exist for Python, C, C++, Java and Go (Martin Abadi et al., 2015; Tensorflow, 2019). In 2017 Keras has gained official support of the Google Brain team as the first high-level Deep Learning Framework, making it effectively Tensorflows main interface, although alternatives exist (Chollet, 2017b).

#### 2.2.5.1.2 Torch/Pytorch

Torch is a Machine Learning library which has been developed by Ronan Collobert, Koray Kavukcuoglu and Clement Farabet and is being maintained by an active community of developers. Torch gained popularity by being the main Machine Learning library being used by Uber, Twitter and Facebook. Facebook's Artificial Intelligence research team developed Pytorch as an API for torch, to simplify its usage. Torch is mainly implemented in C and can be used using C, C++ and Lua, while Pytorch can be used with Python and is implemented in C++ and Python. Both Torch and Pytorch have been published under a BSD License. The advantage of Pytorch is that a regular debugger can be used. The library is well suited for fast prototyping. (Torch Contributors, 2019)

#### 2.2.5.1.3 Theano

Theano is a Python library for mathematical calculations with multi-dimensional arrays. Theano is not inherently a Machine Learning library, but can be used for both Machine Learning and Deep Learning. Theano sees itself as an improved version of the Python library NumPy, which has been used as the foundation for its development. They added further functions to the original numpy library, which also support the rapid development and implementation of Machine Learning Algorithms. Theano has been published under a BSD-License. (Theano Development Team, 2017)

#### 2.2.5.1.4 Apache MxNet

Apache MxNet is a scalable Deep Learning Framework that was developed by the Apache Software Foundation and has been published under an Apache License 2.0. MxNet supports a multitude of different programming language such as Python, C++, Julia, Matlab and many more. The most prominent API is the Python interface Gluon. (Apache Software Foundation, 2019a)

#### 2.2.5.1.5 Deeplearning4J

Deeplearning4J, also known as DL4J and Eclipe Deeplearning4J, is short for Deep Learning for Java. Because this library uses Java, it is often implemented as part of Android applications. It has been published under an Apache License 2 and is being developed by the company Skymind, a software firm based in San Francisco. (Eclipse Deeplearning4J developement team, 2019)

### *2.2.5.2 High-Level Deep Learning Frameworks*

High-level Deep Learning frameworks do not require advanced coding skills and allow the developer to build Deep Learning Networks within a much shorter amount time than low-level Deep Learning frameworks, which make them useful for quick prototyping.

The following table offers a summer over all High-level Deep Learning frameworks

| **Framework** | **Backend** | **API Language** | **License** |
|---|---|---|---|
| Keras | Tensorflow, Theano, CNTK | Python | MIT License |
| Gluon | MXNet | Python | Apache License 2.0 |
| Sonnet | Tensorflow | Python | Apache License 2.0 |
| Swift for Tensorflow | Tensorflow | Swift | Apache License 2.0 |
| PyTorch | Torch | Python | BSD License |

Additional information about each framework can be found within the following chapters.

#### 2.2.5.2.1 Keras

Keras is an Open Source Deep Learning framework in Python. Keras was originally intended for researchers, enabling them to experiment and prototype with Deep Learning with ease and in a user-friendly way. Keras is published under an MIT License. Keras provides pre-defined building blocks based on low-level Deep-Learning

frameworks such as Tensorflow. While Keras is part of Tensorflow as its main API, it sees itself as an API for other low-level frameworks, such as Theano and CNTK. By using this pre-defined building blocks, certain parameters such as the shape of input-tensors in each layer, are being automatically adapted to the model used. (Chollet & others, 2015)

#### 2.2.5.2.2 Gluon

Gluon is an interface for the Apache MXNet framework, which can be manipulated using Python. Gluon is an Open Source library, which is also published under an Apache License 2.0, since it is part of the Apache MxNet project. Like most high-level Deep Learning frameworks Gluon offers building blocks to easily build Neural Networks, without needing technical knowledge, while still maintaining a high performance. (Apache Software Foundation, 2019b)

#### 2.2.5.2.3 Sonnet

Sonnet is an alternative interface for Tensorflow, which can be used with Python. The main principle and main advantage behind Sonnet is that the representation of modules is done in Python objects, instead of just being an interface. This helps if modules need to be changed after construction, without changing their weights. Other than that Sonnet offers, like most other high-level Deep Learning frameworks, an easily understandable way to implement networks. (Reynolds et al., 2017)

#### 2.2.5.2.4 Swift for Tensorflow

Swift for Tensorflow, also called S4TF, is another interface for Tensorflow, which can be accessed by using the programming language Swift. This is developed by the Google Brain team and the reasons for making this framework on top of Keras, is because of the challenges that some Tensorflow developers identified while working with Python. These challenges are performance, concurrency, deployment for mobile as well as the fact that often Python prototypes have to be rewritten in the C++ API for production, and the lack of possibility to write custom operations in Python. (Wei, 2018)

# 3. Methodology

The methodology of this thesis consists of four steps:

1. **Literature review:** The literature review consists of the following steps:
    a. Builiding a basic understanding of Deep Learning.
    b. Identification of the state of the art in text classification with Deep Learning and identification of relevant Deep Neural Network models.
    c. Identification of configurations and architectures for the relevant Deep Neural Network models.
    d. Identification of common problems in the application of Deep Learning and possible ways to circumvent or solve these problems.

    The literature review was conducted using primarily the following sources:

    - **portal.acm.org:** The ACM (Association for Computing Machinery) digital library contains a comprehensive collection of publications focused on the field of computing, including many important publications in the field of Deep Learning.
    - **sciencedirect.com:** Science Direct is the database for scientific publications from the Elsevier publishing company. Science Direct contains books and journals across many scientific fields.
    - **ieeexplore.ieee.org:** IEEE Xplore is a research database for journal articles in the following disciplines: Computer science, electrical engineering and electronics.
2. **Development:** Development of relevant strategies and Deep Neural Network models and configurations for text classification of integrity risks. Additional configurations that are subject to tests are the amount of epochs (iterations through all the training data), the size and quality of the training and testing dataset and the type and size of the Word Embeddings.
3. **Implementation:** Implementation of relevant Deep Neural Network models and configurations and testing of the implemented configurations to improve their performance.

    The implementation of the relevant Deep Neural Networks was done using the following tools:

- **Python, Keras and Tensorflow**: For the implementation of the networks, Keras has been chosen as a high-level Deep Learning, with Tensorflow as a low-level Deep Learning Framework backend. The reason for choosing Tensorflow is because of the big community, which offers technical support as well as many examples, which can be used as a template. Another reason for using Tensorflow, is because its main API is Keras, which can be used with Python. Python is a high level programming language that has been used for both the preprocessing of the input data as well as for the implementation of the Deep Neural Network models. Both Python and Keras made the implementation of the models a comparatively easy task, because its syntax is easy to understand and the building blocks which Keras provide reduce the amount of micromanaging parameters greatly.
- **Jupyter Notebook and Github**: Jupyter Notebook is an interactive web-application which allows to edit, run and order code in a clear and convenient way. It also increases organization by storing outputs and results in the same file as the code. Github has been used to make the final code available to the public.

Additionally the following resources have been used as a basis for the implementation:

- **Keras team Github:** The Keras team published examples of Deep Learning implementations on their Github page (keras-team, 2019). These were used as a template for most of the model implementations. Specifically the examples using the imdb-dataset have been used, since it is also a binary classification task.
- **Kaggle**: Kaggle is an online platform, where researchers and data scientists can upload submissions for machine learning challenges, which are then available to the public. Thus, this platform offers a wide variety of tested and commented code samples, especially for Keras. (Kaggle Inc., 2019)
- **Deep Learning with Python**: The book “Deep Learning with Python” by François Chollet has been used to get to know the basics of Deep Learning and adapt and fine-tune the individual layers in the implementations. (Chollet, 2017a)

4. **Evaluation:** Comparison of the results for each architecture, followed by analysis and discussion of the results. If results do not coincide with what the literature suggests, additional tests will be conducted with different parameters according to the steps mentioned above.

The above-described process is an iterative process. After the results have been compared and analyzed, the process starts over again by implementing the lessons learned in the prior implementation.

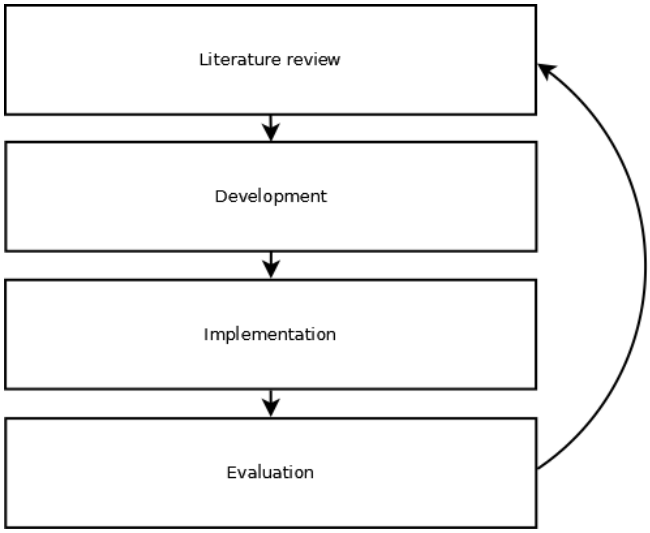



*Figure 11: Vizualization of the research design as an iterative processs*

### 3.1 Description Of The Task

The task is to classify news articles into the following two categories:

1. Integrity-risk news articles
2. Non-integrity-risk news articles

This is a binary classification problem, thus supervised learning will be used to solve the task. The input data consists of text files, thus vectors, one-dimensional tensors, will represent the data. This means that the layers either need to be able to take in vector data or alternatively that the dimensionality of the data needs to change before being processed by the layer, for example by adding a flatten layer. The answers provided to the input data are labels of the two above mentioned categories.

### 3.2 Description of the data & data preprocessing

The input data consists of german text files and the focus is on corruption based integrity risks. It consists of two major datasets, one which was automatically extracted and one that was manually checked. Both datasets have been put together by the Integrity Risk Monitor project team. The following three datasets are a mix of these two datasets and have been used for the benchmark:

**Dataset 1:** This dataset consists of a total of 1385 positive and 1385 negative samples. These samples have been automatically extracted using keyword extraction, thus may contain false-negatives or false-positives as well as unrelated documents. Since the data has been collected by using keyword extraction, the difference between the positive and negative datasets are more obvious and the keywords used for the selection are obvious features, which the networks most likely train to detect.

**Dataset 2:** This dataset consists of a total of 503 positive and 721 negative samples. These samples have been automatically extracted and then manually checked by the project partners, to minimize wrongly classified documents in either of the pools as well as to have a dataset with more similar pools, with ambiguous entries in both, thus making them harder to classify.

**Dataset 3:** This dataset is a mix of the two other datasets, thus contains both ambiguous samples in both the negative and positive dataset, as well as more obvious samples of the first dataset. The composition of this dataset is as follows: 503 positive and 721 negative samples manually picked and 1118 positive and 900 negative samples picked through keyword extraction. There is a total of 1621 positive and 1621 negative samples in this dataset.

Before training the input data is cleaned by using the Python library Natural Language Toolkit, also called NLTK. The aim of data cleaning is to remove noise and errors in the input data. The following steps have been taken as part of the data preprocessing:

- Special characters such as punctuation characters were removed either by just leaving them out or by inserting a space.
- Stopwords have been removed from the custom Word Embedding, because only a pre-defined amount of words are being included in the custom embedding and processed by the Neural Network, which might be problematic if stopwords were included. The reason is because they might

take up a significant amount of the Word Embedding, which are not relevant to the output and could potentially lead to the detection of features, which do not hold any significance. This has not been done for the pre-trained Word Embedding, because stopwords, which are not contained in the pre-trained Word Embedding, are not included and thus get cleaned out. The pre-trained Word Embedding already contains many stopwords, but sets the value of their weights to 0, if they get removed beforehand. To identify stopwords the german NLTK corpus was being used.

## 3.3 Deep Neural Network Models

The result of the implementation are a mixture of the in chapter 4.2 mentioned Deep Neural Network types and layer types. For this thesis, a supervised approach is suitable, since the aim is to categorize a dataset into known and predefined categories. In the following chapters the choice of Word Embeddings and Deep Neural Network models and their layers will be discussed.

### 3.3.1 Word Embeddings

Two different Word Embedding types have been implemented. The first one is a custom Word Embedding, build by using the input text data. This Word Embedding is limited to the top 500 words, which remained after pre-processing the input data. Although further tests with bigger custom Word Embeddings have been conducted, as can be seen in chapter 6.4. The second type uses a pre-trained Word Embedding. The implementation of both the pre-trained and the custom Word Embedding is based on the Github examples of Keras, as well as on a variety of Kaggle entries.

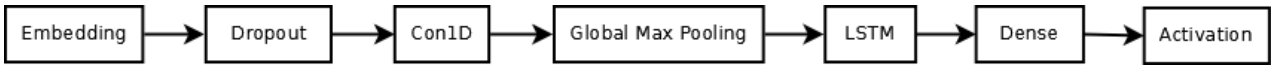
The pre-trained Word Embedding is a German Word2Vec embedding, which has been trained using the German version of Wikipedia as well as a dataset of German news from the years 2007-2013 (Müller, 2019). This Word Embedding has been chosen, because it was trained using the biggest and most up to date corpus of news articles and it is important to have a Word Embedding, which was trained with a corpus similar to the training corpus. The Word Embedding contains 60’000 words and has been converted into text form and into the GloVe format.

### 3.3.2 Network Models

The following two CNN models are based on the models described in the literature review as well as on the examples given by the Keras team. (keras-team, 2019)

CNN: This network model is a standard CNN model as described by Lecun et. al. with just one CNN layer. The CNN layer uses a ReLU activation function, as is the standard for CNN layers in Neural Networks.

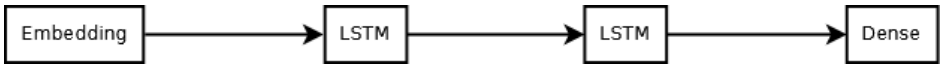

RCNN: This network model is based on the architecture proposed by Li & Wu using an LSTM layer as the recurrent layer.



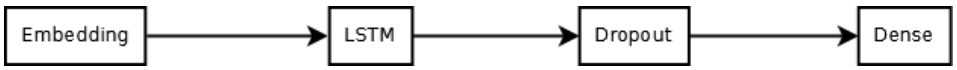

The following RNN models are based on the model architectures proposed by Graves (Graves, 2012, Chapter 5.2) as well as on the example given by the Keras team (keras-team, 2019).

LSTM: For this network, the standard LSTM layer of Keras will be implemented. The LSTM layer is followed by a dense layer as an output layer.


Double LSTM: This network is identical to the regular LSTM model, but with an additional LSTM layer. When stacking LSTM layers on top of each other the parameter return_sequences needs to be set to the value “True”, because else the second LSTM layer does not have a 3-dimensional input. (Chollet & others, 2015)

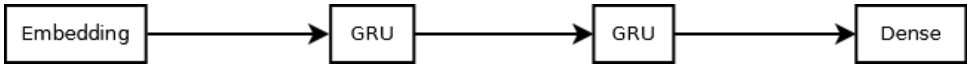



LSTM with dropout layer: The additional dropout layer could potentially allow the network to train for longer, without running into an overfitting problem.

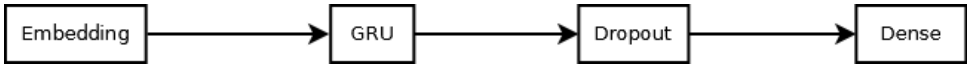



GRU: For this network, the standard GRU layer of Keras was implemented, followed by a dense layer as an output layer.

Double GRU: This network is identical to the regular GRU model, but with an additional LSTM layer. The return sequences parameter needs to be set to true for the first GRU layer, to ensure that the shape of the input for the second GRU layer is correct.

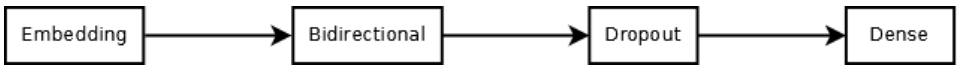



GRU with dropout layer: The added dropout layer is added to this model is to counteract Overfitting and to see, if there is a difference in accuracy with this countermeasure.




RNN/Bidirectional LSTM: This model is identical to the standard LSTM, but with the LSTM being a bidirectional instead of a unidirectional layer.


RNN/Bidirectional LSTM + Dropout: This model is based on the unidirectional LSTM and dropout model, with a bidirectional LSTM layer.




All of the output layers use a Sigmoid activation function, because the task is a binary classification task. All the hidden layers use a ReLU activation function to counteract Overfitting.

# 4. Evaluation

The main problem with the network's performance was the quality of the data and Overfitting. Other potential problems arose because of the Word Embeddings and the model architectures. The following chapters discuss the results of the trainings, the conclusions and where additional research is necessary.

## 4.1 Training Results

The following chapters list the results for the two Word Embedding types described in chapter 5.3.1 and the three datasets described in chapter 5.2. The first value is the name of the model, as described in chapter 5.3.2, the second describes the achieved accuracy score for the evaluation dataset and the third describes the amount of epochs needed to achieve this score, before the output model experienced an obvious overfitting problem or stopped improving. Because Overfitting and the halt of improvement are relative values, which are not precisely defined in the literature, the following values have been set as a measure to define these problems in this thesis:

- As a measure to define a network, which is not improving anymore, a change of less than 0.1% over more than 2 epochs has been chosen.
- An obvious overfitting problem occurs if the training accuracy increases twice as much compared to the training accuracy for more than 1 epoch.

If one of the above conditions is true, the value before the problem occurred is used as an accuracy measure in the following table. The maximum amount of epochs the networks were trained for is 10. The results of each specific epoch of each network training can be found in the appendix.

### 4.1.1 Test Results Custom Word Embeddings

| Model | Result/Test Accuracy | Number of epochs |
|---|---|---|
| **Dataset 1** | | |
| CNN | 0.9762 | 1 |
| RCNN | 0.9866 | 1 |
| LSTM | 0.9783 | 3 |
| Double LSTM | 0.9743 | 1 |
| LSTM + Dropout | 0.9815 | 10 |
| GRU | 0.9865 | 1 |
| Double GRU | 0.9826 | 2 |
| GRU + Dropout | 0.9871 | 5 |
| LSTM bidirectional | 0.9868 | 3 |
| LSTM bidirectional + Dropout | 0.9875 | 5 |
| **Dataset 2** | | |
| CNN | 0.7006 | 2 |
| RCNN | 0.6942 | 4 |
| LSTM | 0.6991 | 3 |
| Double LSTM | 0.7018 | 4 |
| LSTM + Dropout | 0.7004 | 2 |
| GRU | 0.7098 | 3 |
| Double GRU | 0.7029 | 1 |
| GRU + Dropout | 0.7058 | 6 |
| LSTM bidirectional | 0.7001 | 3 |
| LSTM bidirectional + Dropout | 0.6892 | 3 |
| **Dataset 3** | | |
| CNN | 0.7704 | 1 |
| RCNN | 0.7730 | 1 |
| LSTM | 0.7543 | 1 |
| Double LSTM | 0.7485 | 1 |
| LSTM + Dropout | 0.7718 | 6 |
| GRU | 0.7776 | 4 |
| Double GRU | 0.7795 | 7 |
| GRU + Dropout | 0.7778 | 7 |
| LSTM bidirectional | 0.7783 | 6 |
| LSTM bidirectional + Dropout | 0.7611 | 1 |

### 4.1.2 Test Results Pre-Trained Word Embeddings

| Model | Result/Test Accuracy | Number of epochs |
|---|---|---|
| **Dataset 1** | | |
| CNN | 0.9260 | 1 |
| RCNN | 0.9567 | 1 |
| LSTM | 0.9386 | 10 |
| Double LSTM | 0.8736 | 1 |
| LSTM + Dropout | 0.9269 | 4 |
| GRU | 0.7735 | 1 |
| Double GRU | 0.9386 | 6 |
| GRU + Dropout | 0.9025 | 6 |
| LSTM bidirectional | 0.8060 | 1 |
| LSTM bidirectional + Dropout | 0.7924 | 1 |
| **Dataset 2** | | |
| CNN | 0.6906 | 7 |
| RCNN | 0.6516 | 3 |
| LSTM | 0.6783 | 3 |
| Double LSTM | 0.7111 | 3 |
| LSTM + Dropout | 0.6906 | 7 |
| GRU | 0.6414 | 3 |
| Double GRU | 0.6168 | 1 |
| GRU + Dropout | 0.6352 | 1 |
| LSTM bidirectional | 0.7234 | 10 |
| LSTM bidirectional + Dropout | 0.5779 | 1 |
| **Dataset 3** | | |
| CNN | 0.8114 | 1 |
| RCNN | 0.7899 | 1 |
| LSTM | 0.7778 | 7 |
| Double LSTM | 0.8079 | 10 |
| LSTM + Dropout | 0.8171 | 7 |
| GRU | 0.7793 | 2 |
| Double GRU | 0.8295 | 3 |
| GRU + Dropout | 0.9025 | 6 |
| LSTM bidirectional | 0.8171 | 10 |
| LSTM bidirectional + Dropout | 0.7215 | 5 |

### 4.2 Evaluation Of Quality And Sze Of Datasets

The difference in size between the datasets is most probably the biggest influence on the datasets difference in accuracy and also the reason, why dataset 1 is the highest scoring dataset. The quality of dataset 1 is poor due to the way it was extracted. Because all of dataset 1 and a part of dataset 3 have been automatically extracted by using keyword extraction, it is most likely, that the networks identified the keywords used for the extraction as features. The weights of these features are particularly high, because this is a distinctive difference between the two datasets. The aim of the task, is to differentiate between documents, even if certain keywords might be included both in some negative and positive examples, as well as to be able to identify positive documents, even if they do not include certain keywords. The output models for this dataset are most likely not able to do this anymore.

On top of that, the content of some negative example is not news content, which distorts the accuracy further, because the network might learn to differentiate between news and not news documents, instead of integrity and not integrity risks. These distortions are not visible in the accuracy of the test datasets, because they have the same origin and thus have the same quality as the training dataset. This means that the test's accuracy score does not reflect reality, which is most likely the reason why the test accuracy for dataset 1 is that high.

Dataset 2 on the other hand contains an accurate depiction of a potential news corpus containing positive and negative integrity risk samples. Because the dataset has been evaluated manually, the quality of the dataset is much higher and the samples are harder to categorize, due to their selection not being based on certain keywords. In addition, the quality of the negative dataset is much higher, because they were also manually selected, thus negative samples are guaranteed to be news articles. Even though dataset 2 is the dataset with the highest quality, the performance was lower. This was to be expected because of its more ambiguous nature. Also the difference between the different architectures was not significant.

The small size of dataset 2 is most likely the reason for its low accuracy, which could lead to the following problems:

- Smaller datasets are more susceptible to Overfitting. The fact that for dataset 1 the differences were not significantly bigger and the average number of

epochs before Overfitting (6.5 for Dataset 1 and 7 for Dataset 2) were not significantly higher, does not support this theory.

- The size of the dataset does not allow to build a model which has the ability to generalize. This is due to the fact that a dataset this small, cannot contain all possible differences between positive and negative samples, especially considering the fact, that 20% of the dataset is not used to train, because it is kept aside for testing. In a big and diverse dataset, features which are not descriptive for the task, but are prominent in a small number of files in the specific dataset, will be eventually balanced out and thus cleaned out by other examples, which do not have these features. In a small dataset these features have a much bigger weight and not enough samples to balance them out.
- The test accuracy is calculated only on a very small dataset of 20% and thus might be distorted. For 500 positive examples, this is only 100 positive test examples. If the test data is poorly chosen and contains all the samples with a certain feature, which is not present in the training data, it is not possible for the algorithm to learn these features, except by randomly doing so. On the other hand, the training dataset might contain most samples with a certain feature, which might make the output model too biased towards certain features, which are not reflected in the test dataset.

The above-mentioned reasons were the reason why dataset 3 was created. Dataset 3 includes both a big amount of easily distinguishable positives and negatives of dataset 1, as well as the more ambiguous samples found in dataset 2. This offers both the advantages of a big dataset, but also counters some of the disadvantages of the low quality in dataset 1, by adding samples, which are harder to distinguish. While it is most likely still too small to build a model, which can also identify samples from other sources with a high accuracy, it might be big enough to simulate a real dataset and thus to compare the different models with each other.

The following table shows the average accuracy for each dataset:

| | **Custom Word Embedding** | **Pre-Trained Word Embedding** |
|---|---|---|
| **Dataset 1** | 0.9827 | 0.8835 |
| **Dataset 2** | 0.7004 | 0.6617 |
| **Dataset 3** | 0.7692 | 0.8054 |
| **Average** | 0.8175 | 0.7835 |

### 4.3 Evaluation Of Overfitting And Regularization

The CNN and RCNN model started to experience an overfitting problem on average after approximately 2 epochs. The RNN models reached their maximum accuracy before experiencing Overfitting on average after approximately 4 epochs. The evaluation method to identify the accuracy values mentioned in chapter 5.4 was accurate for most of the results, with a few exceptions. For example did the CNN model with dataset 1 grow further even though the difference between the training accuracy and the test accuracy was significant, because the training accuracy started out lower than usual. The following table shows the average amount of epochs a network went through before experiencing Overfitting on average over all datasets and both Word Embedding types.

| **Model** | **Epochs before Overfitting on average / all datasets** | **Epochs before Overfitting on average / dataset 3** |
|---|---|---|
| CNN | 2.2 | 1 |
| RCNN | 1.8 | 1 |
| LSTM | 4.5 | 4 |
| Double LSTM | 3.3 | 5.5 |
| LSTM + Dropout | 6 | 6.5 |
| GRU | 2.3 | 3 |
| Double GRU | 3.8 | 6.5 |
| GRU + Dropout | 5.2 | 6.5 |
| LSTM bidirectional | 5.5 | 8 |
| LSTM bidirectional + Dropout | 2.6 | 3 |

The deeper RNN networks have shown, that there is a significant difference compared to the more shallow and less complex alternative models. The LSTM model showed a decrease on average over all datasets, but an increase for dataset 3. The GRU model increased both overall and for only for dataset 3, by 1.5 respectively 3.5 epochs.

Already an Overfitting problem can be seen in these simple networks, thus it can be assumed, that a more complex or deep network might make this problem worse.

The following regularization steps have been contemplated and applied if they were applicable:

- **Dropout Layers:** Dropout layers did have a significant influence on the amount of epochs, but this only prolonged training for a few more epochs, before facing Overfitting again. The only exception to this were the bidirectional networks, which scored lower with a dropout layer.
- **Decrease of network size and complexity:** The model architectures were already as simple as possible, so the problem is most likely not the amount of parameters.
- **Size and quality of dataset:** As mentioned in the preceding chapter, the datasets were quite small, especially dataset 2. Since even after applying dropout layers Overfitting was a problem, this is the most likely reason why Overfitting occurred. This theory is also supported by the amount of epochs until saturation, which on average were higher in the bigger datasets, except for the CNN networks.
- **Reinitializing weights:** Since Overfitting could just be a product of an unlucky set of starter weights, the networks that experienced the most Overfitting, were reinitialized and computed again. The difference in accuracy was not significant, although in some cases Overfitting set in at a later stage. This is most likely because the initial set of weights gets adjusted quickly and do not have a big influence on their accuracy.

### 4.4 Evaluation Of Word Embeddings

The difference in accuracy between the pre-trained and the custom Word Embedding for dataset 1 and dataset 2 on the other hand is counterintuitive. With dataset 1 the custom Word Embedding achieved considerably higher test accuracy than the pre-trained Word Embedding. Although with dataset 2 the differences are less significant, the pre-trained Word Embedding scored lower than the custom one as well. The average score of each Word Embedding for each Dataset can be seen in the following table.

| | Custom Word Embedding | Pre-Trained Word Embedding |
|---|---|---|
| **Dataset 1** | 0.9827 | 0.8835 |
| **Dataset 2** | 0.7004 | 0.6617 |
| **Dataset 3** | 0.7692 | 0.8054 |
| **Average** | 0.8175 | 0.7835 |

A pre-trained dictionary should be more efficient for smaller datasets, while a custom Word Embedding should be more efficient for bigger datasets. This is because a custom Word Embedding is adapted to a specific dataset, but it can only be specific enough, if the dataset is large enough. A pre-trained Word Embedding on the other hand, was trained without the specific task in mind and thus uses data which might not be representative of the input dataset.

Reasons why the custom Word Embedding performed better in dataset 1 and 2 than the pre-trained Word Embedding could be:

- **The age of the Word Embedding:** The Word Embedding was trained in 2015. Even though ethics and the definition of what an integrity risk is changes over time, the articles focus on corruption, which is neither a new field nor one that changed very much in the last five years. This could be verified or falsified by training a new Word Embedding with up-to-date data, although it would be hard to determine if the differences would be because the data is more up-to-date or just because the data is different.
- **The domain of the Word Embedding:** The Word Embedding does not apply to the task. This is also unlikely, since it was made using news articles among other things. This could be verified by training a new Word Embedding, using the source code provided by the creator, but without including the Wikipedia dataset.
- **Data preprocessing and size:** The custom Word Embedding uses only the top 500 words after data preprocessing, while the pre-trained Word Embedding includes 60’000 words and thus has an input layer of 60’000 neurons. The weight of all the words, which are not part of the input dataset, is set to 0. They are thus ignored, which reduces the size of the network. This and because the Word Embedding itself already serves as a filter, by leaving out words, which are not present in the Word Embedding, is the

reason, why stop word elimination was not done for the pre-trained Word Embedding models. While this is positive, because it ensures, that only words which are part of the Word Embedding end up in the network, thus the network does not train with wrongly spelled words or unimportant names and abbreviations, it also makes the network consider words, which are not as descriptive and less unique such as stop words. This might make the network find features, which are not descriptive for the specific label, but descriptive for the specific dataset and/or author, source or any other characteristic which are not relevant for the detection of integrity risks. This might also be a reason, why dataset 1 has such a high accuracy with a custom Word Embedding. Because the custom Word Embedding only includes the top 500 words, it is most likely, that some of these words are the keywords used to extract the dataset, which are very distinctive features, because they are only included in the positive dataset.

To test this theory tests with a CNN network and a bigger custom Word Embedding have been conducted, which yielded the following results:

| Size of Word Embedding | Result/Test Accuracy with stopword reduction | Result/Test Accuracy without stopword reduction |
|---|---|---|
| 500 | 0.7743 | 0.7701 |
| 1000 | 0.7761 | 0.7836 |
| 1500 | 0.7887 | 0.7852 |
| 2000 | 0.7929 | 0.7829 |
| 2500 | 0.7853 | 0.7896 |
| 3000 | 0.7885 | 0.7831 |
| 3500 | 0.7919 | 0.7891 |
| 4000 | 0.7944 | 0.7963 |

Networks with stopword reduction did not make up for a bigger difference than 1% and in no clear direction, thus did not have a significant impact on the accuracy. The results show a small increase in accuracy in bigger Word Embeddings. The detailed results can be found in the appendix in Chapter 9.1.3.1.

### 4.5 Evaluation Of Model Architecture

The following table shows the average score of each model over all datasets and dataset 3 specifically and both Word Embeddings:

| **Model** | **Average accuracy / all datasets** | **Average accuracy / dataset 3** |
|---|---|---|
| CNN | 0.8125 | 0.7909 |
| RCNN | 0.8086 | 0.7814 |
| LSTM | 0.8044 | 0.7660 |
| Double LSTM | 0.8028 | 0.7782 |
| LSTM + Dropout | 0.8147 | 0.7944 |
| GRU | 0.7780 | 0.7784 |
| Double GRU | 0.8083 | 0.8045 |
| GRU + Dropout | 0.8184 | 0.8401 |
| LSTM bidirectional | 0.8019 | 0.7977 |
| LSTM bidirectional + Dropout | 0.7549 | 0.7413 |

The differences between LSTM and GRU networks were bigger than expected, even though they should achieve similar results according to the literature, with the main difference being their efficiency.

The difference of efficiency turned out to be as expected: GRU was much more efficient than LSTM. It managed to calculate twice the amount of epochs compared to LSTM in all of its occurrences. The millisecond per step value cannot always be compared though, because it depends on circumstances at the time, such as the available computing power, which is distorted by the amount of networks running, as well as other tasks done by the machine at the same time.

There was no significant improvement by stacking two LSTM layers on top of each other, compared to the more shallow and less complex variant with only one LSTM layer. For GRU there was a significant difference between the stacked and non-stacked version, with a difference of approximately 2%. Because the literature suggests that they should have similar scores and the result could be due to an unlucky draft of initial weights, all 4 networks have been computed again 2 times with dataset 3 to get an average score after 4 epochs, as is the average time for an RNN to have an Overfitting problem. For the custom Word Embedding a size of 2000 has been

chosen, to reflect the findings of Chapter 6.4. The results can be seen in the following table:

| | | Accuracy custom Word Embedding | Accuracy pre-trained Word Embedding |
|---|---|---|---|
| **GRU** | First training | 0.7788 | 0.8017 |
| | Second training | 0.7835 | 0.7971 |
| | Average Score | **0.7812** | **0.7994** |
| **Double GRU** | First training | 0.7899 | 0.7894 |
| | Second training | 0.7889 | 0.7840 |
| | Average Score | **0.7894** | **0.7867** |
| **LSTM** | First training | 0.7819 | 0.7662 |
| | Second training | 0.7908 | 0.7986 |
| | Average Score | **0.7864** | **0.7824** |
| **Double LSTM** | First training | 0.7883 | 0.7623 |
| | Second training | 0.7913 | 0.8380 |
| | Average Score | **0.7898** | **0.8002** |

The results show that the difference between the shallow and the deep networks are less than 0.5% on average and not in favor of any of the two configurations and thus not significant. With the bigger custom Word Embedding layer the differences between the two types of Word Embeddings were reduced to approximately 2,2%.

The models which included dropout layers scored higher generally, except for the LSTM bidirectional networks. The difference in RNN models was 2.5% higher, while for the LSTM bidirectional model it was approximately 5% lower. The higher score is because Overfitting was bypassed for a longer time in the models with output layers, which enabled them to train for more epochs as well as to train to detect features, which were less prominent.

The CNN-based networks scored higher than most RNN networks, except the ones with dropout layers, with the exception of the bidirectional model. This is also likely due to the inclusion of the two dropout layers.

The following table shows the average result after 4 trainings of the 4 highest scoring models, CNN, LSTM with a dropout layer, LSTM bidirectional and GRU with a dropout layer. Since pre-trained Word Embeddings scored slightly higher, even in the trainings

with a bigger custom Word Embedding, these implementations were done with a pre-trained Word Embedding. The amount of epochs has been set to the average amount of epochs the respective network managed to train for before facing an overfitting problem as described in Chapter 6.3.

| Model | Attempt | Accuracy |
|---|---|---|
| **CNN** | First training | 0.8588 |
| | Second training | 0.8457 |
| | Third training | 0.8688 |
| | Fourth training | 0.8380 |
| | Average Score | **0.8528** |
| **LSTM + Dropout** | First training | 0.7847 |
| | Second training | 0.8241 |
| | Third training | 0.8256 |
| | Fourth training | 0.8272 |
| | Average Score | **0.8154** |
| **GRU + Dropout** | First training | 0.7878 |
| | Second training | 0.7971 |
| | Third training | 0.8009 |
| | Fourth training | 0.8002 |
| | Average Score | **0.7965** |
| **LSTM bidirectional** | First training | 0.7878 |
| | Second training | 0.7569 |
| | Third training | 0.7878 |
| | Fourth training | 0.8272 |
| | Average Score | **0.7899** |

The highest scoring model is the CNN model with a difference in accuracy of more than 4% and an average accuracy of 85%. To confirm if CNN networks work best for integrity risk detection in general or just for this dataset, additional tests with bigger datasets containing samples with higher quality have to be undertaken.

# 5 Conclusion

To detect integrity risks in news documents is a binary classification task, thus needs a supervised Deep Learning approach. This means that input data with answers to the task needs to be provided. In this case the answers come in form of two labels, integrity risk news articles and not integrity risk news articles. The literature review concluded, that CNN and RNN networks work best for NLP tasks. For the experiments various types of networks based on both CNN and RNN were implemented and evaluated.

The conclusion of the experiments are the following:

- Pre-trained Word Embedding work slightly better for this task with this particular dataset than custom Word Embeddings. The reason for this might be the size of the input data. With a bigger input dataset, a custom Word Embedding should be considered. Bigger custom Word Embeddings had a higher accuracy than smaller ones.
- Dropout layers help to counteract Overfitting, thus allow networks to run for more epochs. This has been shown to lead to better results overall.
- Data quality and size are two of the most important factors to get good results. The bigger the dataset, the more the network can generalize. The better the quality of the dataset, the better it depicts non-simulated data and the less likely the network identifies features, which are non-descriptive of integrity risks.
- In the initial experiments, the CNN models and the LSTM and GRU models with dropout layers, as well as the bidirectional LSTM yielded the best results. Additional tests revealed that the CNN model yields the best results of them with the quality and size of the current dataset.

Deep Learning is a viable option to detect integrity risks in news articles. Both CNN and RNN (LSTM and GRU) as well as Bidirectional RNN networks should be evaluated for an integrity risk classification solution. It is important to build a dataset with high quality, which is big enough to be able to use Deep Learning for this task. To see if Deep Learning performs better than conventional methods for this task additional research is required.

## 6. Literature

Abdel-Hamid, O., Mohamed, A. R., Jiang, H., Deng, L., Penn, G., & Yu, D. (2014). Convolutional neural networks for speech recognition. *IEEE Transactions on Audio, Speech and Language Processing*, *22*(10), 1533–1545. https://doi.org/10.1109/TASLP.2014.2339736

Apache Software Foundation. (2019a). Apache MXNet. Retrieved from https://mxnet.incubator.apache.org/versions/master/index.html

Apache Software Foundation. (2019b). Gluon Package. Retrieved July 25, 2019, from https://mxnet.incubator.apache.org/api/python/gluon/gluon.html

Bengio, Y., Ducharme, R., Vincent, P., & Janvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, *3*, 1137–1155.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*(2), 157–166. https://doi.org/10.1109/72.279181

Carbonell, J. G., Michalski, R. S., & Mitchell, T. M. (1983). Machine Learning: A Historical and Methodological Analysis. *AI Magazine*, *4*(3), 69–79. Retrieved from https://www.aaai.org/ojs/index.php/aimagazine/article/view/406

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. Retrieved from http://arxiv.org/abs/1406.1078

Chollet, F. (2017a). *Deep Learning with Python* (1st ed.). Greenwich, CT, USA: Manning Publications Co.

Chollet, F. (2017b). we will be integrating Keras (TensorFlow-only version) into TensorFlow. Retrieved July 2, 2019, from Twitter website: https://twitter.com/fchollet/status/820746845068505088

Chollet, F., & others. (2015). Keras Documentation. Retrieved May 30, 2019, from https://keras.io

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P. (2011). Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research*, *12*, 2493–2537. https://doi.org/10.1109/CIC.2017.00050

Donahue, J., Hendricks, L. A., Rohrbach, M., Venugopalan, S., Guadarrama, S., Saenko, K., & Darrell, T. (2017). Long-Term Recurrent Convolutional Networks for Visual Recognition and Description. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *39*(4), 677–691. https://doi.org/10.1109/TPAMI.2016.2599174

Eclipse Deeplearning4J developement team. (2019). Deeplearning4j Guide. Retrieved from https://deeplearning4j.org/docs/latest/

Fukushima, K. (1988). Neocognitron: A Hierarchical Neural Network Capable of Visual Pattern Recognition. *Neural Networks*, *1*(2), 119–130.

https://doi.org/https://doi.org/10.1016/0893-6080(88)90014-7

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Journal of Machine Learning Research - Proceedings Track* (Vol. 9).

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks. In G. Gordon, D. Dunson, & M. Dudík (Eds.), *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (pp. 315–323). Retrieved from http://proceedings.mlr.press/v15/glorot11a.html

Gluon Contributors. (2017). Gluon - The Straight Dope 0.1. Retrieved from https://gluon.mxnet.io

Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*. https://doi.org/10.1007/978-3-642-24797-2

Hawkins, D. M. (2004). The Problem of Overfitting. *Journal of Chemical Information and Computer Sciences*, *44*(1), 1–12. https://doi.org/10.1021/ci0342472

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Comput.*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Kaggle Inc. (2019). Kaggle. Retrieved from https://www.kaggle.com/

keras-team. (2019). Keras examples. Retrieved May 30, 2019, from https://github.com/keras-team/keras/tree/master/examples

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*. https://doi.org/10.1201/9781420010749

Lai, S., Xu, L., Liu, K., & Zhao, J. (2015). Recurrent Convolutional Neural Networks for Text Classification. *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2267–2273. Retrieved from http://dl.acm.org/citation.cfm?id=2886521.2886636

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324. https://doi.org/10.1109/5.726791

Li, X., & Wu, X. (2015). Long short-term memory based convolutional recurrent neural networks for large vocabulary speech recognition. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, *2015–Janua*, 3219–3223.

Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig~Citro, … Xiaoqiang Zheng. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Retrieved from https://www.tensorflow.org/

McCulloch, W. S., & Pitts, W. (1988). A logical calculus of the ideas immanent in nervous activity. In J. A. Anderson & E. Rosenfeld (Eds.), *Neurocomputing: Foundations of Research* (pp. 15–27). Retrieved from http://dl.acm.org/citation.cfm?id=65669.104377

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. *Advances in Neural Information Processing Systems*, (26), 3111–3119.

Ming Liang, & Xiaolin Hu. (2015). Recurrent convolutional neural network for object recognition. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 3367–3375. https://doi.org/10.1109/CVPR.2015.7298958

Molina, A. D. (2018). A Systems Approach to Managing Organizational Integrity Risks: Lessons From the 2014 Veterans Affairs Waitlist Scandal. *The American Review of Public Administration*, *48*(8), 872–885. https://doi.org/10.1177/0275074018755006

Müller, A. (2019). German word embeddings. Retrieved July 24, 2019, from https://devmount.github.io/GermanWordEmbeddings/

Ng, A. Y. (2018). Machine Learning Yearning: Technical Strategy for AI Engineers, in the Era of Deep Learning. *Deeplearning.Ai*. Retrieved from https://www.deeplearning.ai/content/uploads/2018/09/Ng-MLY01-12.pdf

Nwankpa, C., Ijomah, W., Gachagan, A., & Marshall, S. (2018). *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*. 1–20. Retrieved from http://arxiv.org/abs/1811.03378

Park, S., & Kwak, N. (2017). Analysis on the Dropout Effect in Convolutional Neural Networks. In S.-H. Lai, V. Lepetit, K. Nishino, & Y. Sato (Eds.), *Computer Vision -- ACCV 2016* (pp. 189–204). Cham: Springer International Publishing.

Pennington, J., Socher, R., & Manning, C. D. (2014). GloVe: Global Vectors for Word Representation. *Empirical Methods in Natural Language Processing (EMNLP)*, 1532–1543. Retrieved from http://www.aclweb.org/anthology/D14-1162

Pouyanfar, S., Sadiq, S., Yan, Y., Tian, H., Tao, Y., Reyes, M. P., … Iyengar, S. S. (2018). A Survey on Deep Learning. *ACM Computing Surveys*, *51*(5), 1–36. https://doi.org/10.1145/3234150

Reynolds, M., Barth-Maron, G., Besse, F., de Las Casas, D., Fidjeland, A., Green, T., … Viola, F. (2017). *Open sourcing Sonnet - a new library for constructing neural networks*.

Schmidhuber, J. (2015). Deep Learning in neural networks: An overview. *Neural Networks*, *61*, 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

Schuster, M., & Paliwal, K. K. (1997). Bidirectional Recurrent Neural Networks. *Trans. Sig. Proc.*, *45*(11), 2673–2681. https://doi.org/10.1109/78.650093

Socher, R., Lin, C. C., Ng, A. Y., & Manning, C. D. (2011). Parsing Natural Scenes and Natural Language with Recursive Neural Networks Richard. *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, *11*, 129–136. https://doi.org/10.1007/978-3-540-87479-9

Socher, R., Perelygin, A., Y.Wu, J., Chuang, J., Manning, C. D., Ng, A. Y., & Potts, C. (2013). Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. *PLoS ONE*, *8*(9), 1631–1642. https://doi.org/10.1371/journal.pone.0073791

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. In *Journal of Machine Learning Research* (Vol. 15).

Tensorflow. (2019). Tensorflow Github. Retrieved July 2, 2019, from Github website: https://github.com/tensorflow/tensorflow

Theano Development Team. (2017). Theano 1.0 release. Retrieved July 25, 2019, from http://deeplearning.net/software/theano/

Torch Contributors. (2019). PyTorch Documentation. Retrieved from pytorch.org website: https://pytorch.org/docs/stable/index.html

Wei, R. (2018). Why Swift for TensorFlow? Retrieved July 25, 2019, from https://github.com/tensorflow/swift/blob/master/docs/WhySwiftForTensorFlow.md

Zaremba, W., Sutskever, I., & Vinyals, O. (2014). *Recurrent Neural Network Regularization*. Retrieved from https://arxiv.org/abs/1409.2329

# Appendix

The following chapters contain the detailed training results for each epoch of each training divided into the two Word Embedding types and the three different datasets. An example for each model implementation and each Word Embedding can be found in the following GitHub repository: https://github.com/ukalb/IntegrityRiskDL

## Appendix A Training Results for Custom Word Embedding Models

*CNN*

**Dataset 1:**
Train on 41960 samples, validate on 10490 samples
Epoch 1/4 - 41960/41960 - 292s 7ms/step - loss: 0.0795 - acc: 0.9772 - val_loss: 0.0667 - val_acc: 0.9762
Epoch 2/4 - 41960/41960 - 283s 7ms/step - loss: 0.0443 - acc: 0.9826 - val_loss: 0.0578 - val_acc: 0.9773
Epoch 3/4 - 41960/41960 - 302s 7ms/step - loss: 0.0386 - acc: 0.9839 - val_loss: 0.0633 - val_acc: 0.9772
Epoch 4/4 - 41960/41960 - 293s 7ms/step - loss: 0.0352 - acc: 0.9854 - val_loss: 0.0612 - val_acc: 0.9787
10490/10490 - 18s 2ms/step
Test score: 0.06115965358628843
Test accuracy: 0.978741658722593

**Dataset 2:**
Train on 21040 samples, validate on 5261 samples
Epoch 1/4 - 21040/21040 - 54s 3ms/step - loss: 0.6120 - acc: 0.6900 - val_loss: 0.6085 - val_acc: 0.6932
Epoch 2/4 - 21040/21040 - 59s 3ms/step - loss: 0.5919 - acc: 0.7048 - val_loss: 0.5926 - val_acc: 0.7006
Epoch 3/4 - 21040/21040 - 58s 3ms/step - loss: 0.5834 - acc: 0.7095 - val_loss: 0.5988 - val_acc: 0.6974
Epoch 4/4 - 21040/21040 - 85s 4ms/step - loss: 0.5752 - acc: 0.7138 - val_loss: 0.6056 - val_acc: 0.6892
5261/5261 - 5s 912us/step
Test score: 0.6055545946691137
Test accuracy: 0.6892225812923045

**Dataset 3:**
Train on 58620 samples, validate on 14655 samples
Epoch 1/4 - 58620/58620 - 174s 3ms/step - loss: 0.5208 - acc: 0.7648 - val_loss: 0.5139 - val_acc: 0.7704
Epoch 2/4 - 58620/58620 - 134s 2ms/step - loss: 0.5004 - acc: 0.7766 - val_loss: 0.5061 - val_acc: 0.7716
Epoch 3/4 - 58620/58620 - 109s 2ms/step - loss: 0.4943 - acc: 0.7798 - val_loss: 0.5044 - val_acc: 0.7744
Epoch 4/4 - 58620/58620 - 89s 2ms/step - loss: 0.4882 - acc: 0.7833 - val_loss: 0.5048 - val_acc: 0.7733
14655/14655 - 5s 354us/step
Test score: 0.5047620832208316
Test accuracy: 0.7732514500292605

*RCNN*

**Dataset 1:**
Train on 41960 samples, validate on 10490 samples
Epoch 1/4
41960/41960 - 206s 5ms/step - loss: 0.0742 - acc: 0.9790 - val_loss: 0.0395 - val_acc: 0.9866
Epoch 2/4
41960/41960 - 203s 5ms/step - loss: 0.0352 - acc: 0.9882 - val_loss: 0.0371 - val_acc: 0.9855
Epoch 3/4
41960/41960 - 202s 5ms/step - loss: 0.0303 - acc: 0.9892 - val_loss: 0.0396 - val_acc: 0.9849
Epoch 4/4
41960/41960 - 203s 5ms/step - loss: 0.0280 - acc: 0.9893 - val_loss: 0.0409 - val_acc: 0.9848
10490/10490 - 8s 769us/step
Test score: 0.04094634840684522
Test accuracy: 0.9848426991809085

**Dataset 2:**
Train on 21040 samples, validate on 5261 samples
Epoch 1/4 - 21040/21040 - 50s 2ms/step - loss: 0.6134 - acc: 0.6917 - val_loss: 0.6106 - val_acc: 0.6917
Epoch 2/4 - 21040/21040 - 50s 2ms/step - loss: 0.5917 - acc: 0.7051 - val_loss: 0.5925 - val_acc: 0.7023
Epoch 3/4 - 21040/21040 - 51s 2ms/step - loss: 0.5823 - acc: 0.7120 - val_loss: 0.5952 - val_acc: 0.7023
Epoch 4/4 - 21040/21040 - 50s 2ms/step - loss: 0.5719 - acc: 0.7172 - val_loss: 0.5978 - val_acc: 0.6942
5261/5261 - 2s 351us/step
Test score: 0.5978319810225786
Test accuracy: 0.6941646097209875

**Dataset 3:**
Train on 58620 samples, validate on 14655 samples
Epoch 1/4 - 58620/58620 - 142s 2ms/step - loss: 0.5194 - acc: 0.7663 - val_loss: 0.5019 - val_acc: 0.7730
Epoch 2/4 - 58620/58620 - 117s 2ms/step - loss: 0.4931 - acc: 0.7804 - val_loss: 0.5006 - val_acc: 0.7758
Epoch 3/4 - 58620/58620 - 197s 3ms/step - loss: 0.4841 - acc: 0.7859 - val_loss: 0.4976 - val_acc: 0.7781
Epoch 4/4 - 58620/58620 - 231s 4ms/step - loss: 0.4777 - acc: 0.7902 - val_loss: 0.4990 - val_acc: 0.7762
14655/14655 - 9s 586us/step
Test score: 0.4989652040975975
Test accuracy: 0.7761856010650783

*LSTM*

**Dataset 1:**
Train on 41960 samples, validate on 10490 samples

Epoch 1/10 - 41960/41960 - 681s 16ms/step - loss: 0.1301 - acc: 0.9724 - val_loss: 0.1224 - val_acc: 0.9724
Epoch 2/10 - 41960/41960 - 681s 16ms/step - loss: 0.1026 - acc: 0.9735 - val_loss: 0.0962 - val_acc: 0.9724
Epoch 3/10 - 41960/41960 - 678s 16ms/step - loss: 0.0729 - acc: 0.9774 - val_loss: 0.0653 - val_acc: 0.9783
Epoch 4/10 - 41960/41960 - 681s 16ms/step - loss: 0.1009 - acc: 0.9754 - val_loss: 0.0647 - val_acc: 0.9765
Epoch 5/10 - 41960/41960 - 849s 20ms/step - loss: 0.1049 - acc: 0.9734 - val_loss: 0.1111 - val_acc: 0.9725
Epoch 6/10 - 41960/41960 - 892s 21ms/step - loss: 0.0836 - acc: 0.9742 - val_loss: 0.0757 - val_acc: 0.9742
Epoch 7/10 - 41960/41960 - 1000s 24ms/step - loss: 0.0566 - acc: 0.9788 - val_loss: 0.0557 - val_acc: 0.9802
Epoch 8/10 - 41960/41960 - 1044s 25ms/step - loss: 0.0395 - acc: 0.9857 - val_loss: 0.0531 - val_acc: 0.9838
Epoch 9/10 - 41960/41960 - 994s 24ms/step - loss: 0.0354 - acc: 0.9872 - val_loss: 0.0489 - val_acc: 0.9837
Epoch 10/10 - 41960/41960 - 1215s 29ms/step - loss: 0.0328 - acc: 0.9885 - val_loss: 0.0640 - val_acc: 0.9821

**Dataset 2:**
Train on 21040 samples, validate on 5261 samples
Epoch 1/8 - 21040/21040 - 126s 6ms/step - loss: 0.6246 - acc: 0.6819 - val_loss: 0.6201 - val_acc: 0.6805
Epoch 2/8 - 21040/21040 - 195s 9ms/step - loss: 0.6069 - acc: 0.6949 - val_loss: 0.6068 - val_acc: 0.6919
Epoch 3/8 - 21040/21040 - 268s 13ms/step - loss: 0.5934 - acc: 0.7068 - val_loss: 0.6022 - val_acc: 0.6991
Epoch 4/8 - 21040/21040 - 270s 13ms/step - loss: 0.5897 - acc: 0.7109 - val_loss: 0.6029 - val_acc: 0.6982
Epoch 5/8 - 21040/21040 - 269s 13ms/step - loss: 0.5863 - acc: 0.7115 - val_loss: 0.6041 - val_acc: 0.6980
Epoch 6/8 - 21040/21040 - 269s 13ms/step - loss: 0.5844 - acc: 0.7134 - val_loss: 0.6051 - val_acc: 0.6953
Epoch 7/8 - 21040/21040 - 269s 13ms/step - loss: 0.5821 - acc: 0.7145 - val_loss: 0.6125 - val_acc: 0.6995
Epoch 8/8 - 21040/21040 - 270s 13ms/step - loss: 0.5773 - acc: 0.7173 - val_loss: 0.6134 - val_acc: 0.6974

**Dataset 3:**
Train on 58620 samples, validate on 14655 samples
Epoch 1/10 - 58620/58620 - 1052s 18ms/step - loss: 0.5460 - acc: 0.7498 - val_loss: 0.5270 - val_acc: 0.7543
Epoch 2/10 - 58620/58620 - 570s 10ms/step - loss: 0.5160 - acc: 0.7665 - val_loss: 0.5363 - val_acc: 0.7608
Epoch 3/10 - 58620/58620 - 571s 10ms/step - loss: 0.5134 - acc: 0.7624 - val_loss: 0.5214 - val_acc: 0.7487
Epoch 4/10 - 58620/58620 - 571s 10ms/step - loss: 0.5092 - acc: 0.7666 - val_loss: 0.5086 - val_acc: 0.7681

Epoch 5/10 - 58620/58620 - 571s 10ms/step - loss: 0.4980 - acc: 0.7775 - val_loss: 0.5107 - val_acc: 0.7684
Epoch 6/10 - 58620/58620 - 568s 10ms/step - loss: 0.4990 - acc: 0.7733 - val_loss: 0.5136 - val_acc: 0.7597
Epoch 7/10 - 58620/58620 - 570s 10ms/step - loss: 0.5018 - acc: 0.7704 - val_loss: 0.5373 - val_acc: 0.7415
Epoch 8/10 - 58620/58620 - 568s 10ms/step - loss: 0.4987 - acc: 0.7735 - val_loss: 0.5079 - val_acc: 0.7705
Epoch 9/10 - 58620/58620 - 571s 10ms/step - loss: 0.4886 - acc: 0.7842 - val_loss: 0.5067 - val_acc: 0.7703
Epoch 10/10 - 58620/58620 - 567s 10ms/step - loss: 0.4860 - acc: 0.7854 - val_loss: 0.5033 - val_acc: 0.7750

*Double LSTM*

**Dataset 1:**
Train on 41960 samples, validate on 10490 samples
Epoch 1/10 - 41960/41960 - 1385s 33ms/step - loss: 0.1165 - acc: 0.9727 - val_loss: 0.1468 - val_acc: 0.9743
Epoch 2/10 - 41960/41960 - 1432s 34ms/step - loss: 0.1153 - acc: 0.9739 - val_loss: 0.1196 - val_acc: 0.9743
Epoch 3/10 - 41960/41960 - 1817s 43ms/step - loss: 0.1228 - acc: 0.9735 - val_loss: 0.1197 - val_acc: 0.9743
Epoch 4/10 - 41960/41960 - 2186s 52ms/step - loss: 0.1225 - acc: 0.9735 - val_loss: 0.1200 - val_acc: 0.9743
Epoch 5/10 - 41960/41960 - 2339s 56ms/step - loss: 0.1223 - acc: 0.9735 - val_loss: 0.1188 - val_acc: 0.9743
Epoch 6/10 - 41960/41960 - 1462s 35ms/step - loss: 0.1177 - acc: 0.9735 - val_loss: 0.0976 - val_acc: 0.9742
Epoch 7/10 - 41960/41960 - 1951s 47ms/step - loss: 0.0973 - acc: 0.9735 - val_loss: 0.0818 - val_acc: 0.9745
Epoch 8/10 - 41960/41960 - 1164s 28ms/step - loss: 0.0794 - acc: 0.9739 - val_loss: 0.0815 - val_acc: 0.9744
Epoch 9/10 - 41960/41960 - 886s 21ms/step - loss: 0.0724 - acc: 0.9770 - val_loss: 0.0692 - val_acc: 0.9797
Epoch 10/10 - 41960/41960 - 1180s 28ms/step - loss: 0.0671 - acc: 0.9770 - val_loss: 0.0717 - val_acc: 0.9806

**Dataset 2:**
Train on 21040 samples, validate on 5261 samples
Epoch 1/10 - 21040/21040 - 437s 21ms/step - loss: 0.6212 - acc: 0.6850 - val_loss: 0.6112 - val_acc: 0.6936
Epoch 2/10 - 21040/21040 - 382s 18ms/step - loss: 0.5993 - acc: 0.7019 - val_loss: 0.6059 - val_acc: 0.6972
Epoch 3/10 - 21040/21040 - 237s 11ms/step - loss: 0.5931 - acc: 0.7058 - val_loss: 0.6144 - val_acc: 0.6972
Epoch 4/10 - 21040/21040 - 252s 12ms/step - loss: 0.5872 - acc: 0.7077 - val_loss: 0.6053 - val_acc: 0.7018
Epoch 5/10 - 21040/21040 - 265s 13ms/step - loss: 0.5833 - acc: 0.7117 - val_loss: 0.6038 - val_acc: 0.7029
Epoch 6/10 - 21040/21040 - 279s 13ms/step - loss: 0.5796 - acc: 0.7135 - val_loss: 0.6049 - val_acc: 0.6999

Epoch 7/10 - 21040/21040 - 291s 14ms/step - loss: 0.5763 - acc: 0.7152 - val_loss: 0.6083 - val_acc: 0.6938
Epoch 8/10 - 21040/21040 - 287s 14ms/step - loss: 0.5796 - acc: 0.7141 - val_loss: 0.6048 - val_acc: 0.7001
Epoch 9/10 - 21040/21040 - 234s 11ms/step - loss: 0.5729 - acc: 0.7162 - val_loss: 0.6113 - val_acc: 0.7001
Epoch 10/10 - 21040/21040 - 234s 11ms/step - loss: 0.5698 - acc: 0.7191 - val_loss: 0.6062 - val_acc: 0.7029

**Dataset 3:**
Train on 58620 samples, validate on 14655 samples
Epoch 1/10 - 58620/58620 - 1651s 28ms/step - loss: 0.5454 - acc: 0.7509 - val_loss: 0.5363 - val_acc: 0.7485
Epoch 2/10 - 58620/58620 - 1146s 20ms/step - loss: 0.5341 - acc: 0.7548 - val_loss: 0.5742 - val_acc: 0.7387
Epoch 3/10 - 58620/58620 - 1144s 20ms/step - loss: 0.5635 - acc: 0.7441 - val_loss: 0.5538 - val_acc: 0.7387
Epoch 4/10 - 58620/58620 - 1143s 19ms/step - loss: 0.5559 - acc: 0.7441 - val_loss: 0.5475 - val_acc: 0.7387
Epoch 5/10 - 58620/58620 - 1130s 19ms/step - loss: 0.5235 - acc: 0.7598 - val_loss: 0.5298 - val_acc: 0.7636
Epoch 6/10 - 58620/58620 - 653s 11ms/step - loss: 0.5067 - acc: 0.7750 - val_loss: 0.5179 - val_acc: 0.7678
Epoch 7/10 - 58620/58620 - 654s 11ms/step - loss: 0.5005 - acc: 0.7785 - val_loss: 0.5129 - val_acc: 0.7680
Epoch 8/10 - 58620/58620 - 654s 11ms/step - loss: 0.4999 - acc: 0.7796 - val_loss: 0.5137 - val_acc: 0.7691
Epoch 9/10 - 58620/58620 - 653s 11ms/step - loss: 0.4944 - acc: 0.7821 - val_loss: 0.5133 - val_acc: 0.7672
Epoch 10/10 - 58620/58620 - 664s 11ms/step - loss: 0.4919 - acc: 0.7832 - val_loss: 0.5109 - val_acc: 0.7675

*LSTM + Dropout*

**Dataset 1:**
Train on 41960 samples, validate on 10490 samples
Epoch 1/10 - 41960/41960 - 1005s 24ms/step - loss: 0.1297 - acc: 0.9721 - val_loss: 0.0959 - val_acc: 0.9739
Epoch 2/10 - 41960/41960 - 851s 20ms/step - loss: 0.0902 - acc: 0.9740 - val_loss: 0.1095 - val_acc: 0.9770
Epoch 3/10 - 41960/41960 - 1102s 26ms/step - loss: 0.0671 - acc: 0.9777 - val_loss: 0.0827 - val_acc: 0.9790
Epoch 4/10 - 41960/41960 - 981s 23ms/step - loss: 0.0647 - acc: 0.9806 - val_loss: 0.0842 - val_acc: 0.9745
Epoch 5/10 - 41960/41960 - 1142s 27ms/step - loss: 0.0552 - acc: 0.9820 - val_loss: 0.0577 - val_acc: 0.9826
Epoch 6/10 - 41960/41960 - 1085s 26ms/step - loss: 0.0844 - acc: 0.9771 - val_loss: 0.0596 - val_acc: 0.9807
Epoch 7/10 - 41960/41960 - 727s 17ms/step - loss: 0.0712 - acc: 0.9752 - val_loss: 0.0665 - val_acc: 0.9800
Epoch 8/10 - 41960/41960 - 752s 18ms/step - loss: 0.0484 - acc: 0.9832 - val_loss: 0.0552 - val_acc: 0.9828

Epoch 9/10 - 41960/41960 - 777s 19ms/step - loss: 0.0433 - acc: 0.9847 - val_loss: 0.0481 - val_acc: 0.9837
Epoch 10/10 - 41960/41960 - 806s 19ms/step - loss: 0.0441 - acc: 0.9844 - val_loss: 0.0522 - val_acc: 0.9815

**Dataset 2:**
Train on 21040 samples, validate on 5261 samples
Epoch 1/8 - 21040/21040 - 136s 6ms/step - loss: 0.6257 - acc: 0.6809 - val_loss: 0.6191 - val_acc: 0.6828
Epoch 2/8 - 21040/21040 - 299s 14ms/step - loss: 0.6063 - acc: 0.6943 - val_loss: 0.6022 - val_acc: 0.7004
Epoch 3/8 - 21040/21040 - 431s 21ms/step - loss: 0.5957 - acc: 0.7048 - val_loss: 0.6048 - val_acc: 0.6980
Epoch 4/8 - 21040/21040 - 429s 20ms/step - loss: 0.5907 - acc: 0.7080 - val_loss: 0.6166 - val_acc: 0.6911
Epoch 5/8 - 21040/21040 - 422s 20ms/step - loss: 0.5888 - acc: 0.7096 - val_loss: 0.6005 - val_acc: 0.7001
Epoch 6/8 - 21040/21040 - 210s 10ms/step - loss: 0.5868 - acc: 0.7117 - val_loss: 0.6003 - val_acc: 0.7020
Epoch 7/8 - 21040/21040 - 192s 9ms/step - loss: 0.5852 - acc: 0.7139 - val_loss: 0.5993 - val_acc: 0.7014
Epoch 8/8 - 21040/21040 - 192s 9ms/step - loss: 0.5875 - acc: 0.7116 - val_loss: 0.6006 - val_acc: 0.6997

**Dataset 3:**
Train on 58620 samples, validate on 14655 samples
Epoch 1/10 - 58620/58620 - 671s 11ms/step - loss: 0.5424 - acc: 0.7522 - val_loss: 0.5178 - val_acc: 0.7632
Epoch 2/10 - 58620/58620 - 304s 5ms/step - loss: 0.5150 - acc: 0.7684 - val_loss: 0.5161 - val_acc: 0.7651
Epoch 3/10 - 58620/58620 - 303s 5ms/step - loss: 0.5145 - acc: 0.7670 - val_loss: 0.5157 - val_acc: 0.7690
Epoch 4/10 - 58620/58620 - 303s 5ms/step - loss: 0.5024 - acc: 0.7763 - val_loss: 0.5167 - val_acc: 0.7529
Epoch 5/10 - 58620/58620 - 303s 5ms/step - loss: 0.5068 - acc: 0.7661 - val_loss: 0.5084 - val_acc: 0.7671
Epoch 6/10 - 58620/58620 - 304s 5ms/step - loss: 0.5083 - acc: 0.7654 - val_loss: 0.5058 - val_acc: 0.7718
Epoch 7/10 - 58620/58620 - 303s 5ms/step - loss: 0.5008 - acc: 0.7744 - val_loss: 0.5060 - val_acc: 0.7709
Epoch 8/10 - 58620/58620 - 303s 5ms/step - loss: 0.4985 - acc: 0.7749 - val_loss: 0.5200 - val_acc: 0.7554
Epoch 9/10 - 58620/58620 - 304s 5ms/step - loss: 0.4906 - acc: 0.7813 - val_loss: 0.5062 - val_acc: 0.7713
Epoch 10/10 - 58620/58620 - 304s 5ms/step - loss: 0.4921 - acc: 0.7813 - val_loss: 0.5033 - val_acc: 0.7715

*GRU*

**Dataset 1:**
Train on 41960 samples, validate on 10490 samples

Epoch 1/10 - 41960/41960 - 331s 8ms/step - loss: 0.0418 - acc: 0.9859 - val_loss: 0.0416 - val_acc: 0.9865
Epoch 2/10 - 41960/41960 - 324s 8ms/step - loss: 0.0317 - acc: 0.9886 - val_loss: 0.0405 - val_acc: 0.9861
Epoch 3/10 - 41960/41960 - 325s 8ms/step - loss: 0.0281 - acc: 0.9898 - val_loss: 0.0425 - val_acc: 0.9849
Epoch 4/10 - 41960/41960 - 324s 8ms/step - loss: 0.0263 - acc: 0.9900 - val_loss: 0.0415 - val_acc: 0.9867
Epoch 5/10 - 41960/41960 - 324s 8ms/step - loss: 0.0246 - acc: 0.9904 - val_loss: 0.0466 - val_acc: 0.9859
Epoch 6/10 - 41960/41960 - 324s 8ms/step - loss: 0.0240 - acc: 0.9909 - val_loss: 0.0438 - val_acc: 0.9861
Epoch 7/10 - 41960/41960 - 325s 8ms/step - loss: 0.0225 - acc: 0.9917 - val_loss: 0.0470 - val_acc: 0.9861
Epoch 8/10 - 41960/41960 - 329s 8ms/step - loss: 0.0219 - acc: 0.9919 - val_loss: 0.0461 - val_acc: 0.9862
Epoch 9/10 - 41960/41960 - 325s 8ms/step - loss: 0.0198 - acc: 0.9927 - val_loss: 0.0516 - val_acc: 0.9845
Epoch 10/10 - 41960/41960 - 324s 8ms/step - loss: 0.0188 - acc: 0.9930 - val_loss: 0.0474 - val_acc: 0.9852

**Dataset 2:**
Train on 21040 samples, validate on 5261 samples
Epoch 1/15 - 21040/21040 - 172s 8ms/step - loss: 0.6261 - acc: 0.6798 - val_loss: 0.6191 - val_acc: 0.6894
Epoch 2/15 - 21040/21040 - 219s 10ms/step - loss: 0.6081 - acc: 0.6944 - val_loss: 0.6055 - val_acc: 0.6976
Epoch 3/15 - 21040/21040 - 221s 11ms/step - loss: 0.5965 - acc: 0.7013 - val_loss: 0.5936 - val_acc: 0.7098
Epoch 4/15 - 21040/21040 - 223s 11ms/step - loss: 0.5894 - acc: 0.7058 - val_loss: 0.5940 - val_acc: 0.7059
Epoch 5/15 - 21040/21040 - 222s 11ms/step - loss: 0.5842 - acc: 0.7063 - val_loss: 0.5962 - val_acc: 0.7023
Epoch 6/15 - 21040/21040 - 222s 11ms/step - loss: 0.5803 - acc: 0.7083 - val_loss: 0.5989 - val_acc: 0.7039
Epoch 7/15 - 21040/21040 - 222s 11ms/step - loss: 0.5766 - acc: 0.7109 - val_loss: 0.6023 - val_acc: 0.6997
Epoch 8/15 - 21040/21040 - 222s 11ms/step - loss: 0.5741 - acc: 0.7117 - val_loss: 0.6012 - val_acc: 0.7004
Epoch 9/15 - 21040/21040 - 183s 9ms/step - loss: 0.5721 - acc: 0.7120 - val_loss: 0.6031 - val_acc: 0.7042
Epoch 10/15 - 21040/21040 - 163s 8ms/step - loss: 0.5683 - acc: 0.7134 - val_loss: 0.6065 - val_acc: 0.7073
Epoch 11/15 - 21040/21040 - 164s 8ms/step - loss: 0.5643 - acc: 0.7175 - val_loss: 0.6094 - val_acc: 0.6972
Epoch 12/15 - 21040/21040 - 163s 8ms/step - loss: 0.5602 - acc: 0.7198 - val_loss: 0.6152 - val_acc: 0.7023
Epoch 13/15 - 21040/21040 - 164s 8ms/step - loss: 0.5605 - acc: 0.7187 - val_loss: 0.6160 - val_acc: 0.6997
Epoch 14/15 - 21040/21040 - 163s 8ms/step - loss: 0.5550 - acc: 0.7231 - val_loss: 0.6244 - val_acc: 0.7006

Epoch 15/15 - 21040/21040 - 164s 8ms/step - loss: 0.5521 - acc: 0.7255 - val_loss: 0.6258 - val_acc: 0.7016

**Dataset 3:**
Train on 58620 samples, validate on 14655 samples
Epoch 1/15 - 58620/58620 - 1015s 17ms/step - loss: 0.5562 - acc: 0.7447 - val_loss: 0.5247 - val_acc: 0.7564
Epoch 2/15 - 58620/58620 - 929s 16ms/step - loss: 0.5149 - acc: 0.7680 - val_loss: 0.5111 - val_acc: 0.7679
Epoch 3/15 - 58620/58620 - 933s 16ms/step - loss: 0.5045 - acc: 0.7751 - val_loss: 0.5067 - val_acc: 0.7729
Epoch 4/15 - 58620/58620 - 977s 17ms/step - loss: 0.4977 - acc: 0.7800 - val_loss: 0.4993 - val_acc: 0.7776
Epoch 5/15 - 58620/58620 - 822s 14ms/step - loss: 0.4934 - acc: 0.7824 - val_loss: 0.4977 - val_acc: 0.7772
Epoch 6/15 - 58620/58620 - 866s 15ms/step - loss: 0.4913 - acc: 0.7832 - val_loss: 0.5010 - val_acc: 0.7722
Epoch 7/15 - 58620/58620 - 736s 13ms/step - loss: 0.4894 - acc: 0.7835 - val_loss: 0.4993 - val_acc: 0.7779
Epoch 8/15 - 58620/58620 - 437s 7ms/step - loss: 0.4851 - acc: 0.7871 - val_loss: 0.4989 - val_acc: 0.7769
Epoch 9/15 - 58620/58620 - 558s 10ms/step - loss: 0.4823 - acc: 0.7886 - val_loss: 0.4971 - val_acc: 0.7780
Epoch 10/15 - 58620/58620 - 560s 10ms/step - loss: 0.4787 - acc: 0.7911 - val_loss: 0.4979 - val_acc: 0.7774
Epoch 11/15 - 58620/58620 - 479s 8ms/step - loss: 0.4753 - acc: 0.7923 - val_loss: 0.4997 - val_acc: 0.7756
Epoch 12/15 - 58620/58620 - 534s 9ms/step - loss: 0.4724 - acc: 0.7936 - val_loss: 0.5027 - val_acc: 0.7750
Epoch 13/15 - 58620/58620 - 321s 5ms/step - loss: 0.4682 - acc: 0.7968 - val_loss: 0.5055 - val_acc: 0.7776
Epoch 14/15 - 58620/58620 - 320s 5ms/step - loss: 0.4641 - acc: 0.7984 - val_loss: 0.5150 - val_acc: 0.7732
Epoch 15/15 - 58620/58620 - 320s 5ms/step - loss: 0.4607 - acc: 0.8004 - val_loss: 0.5123 - val_acc: 0.7737

*Double GRU*

**Dataset 1:**
Train on 41960 samples, validate on 10490 samples
Epoch 1/10 - 41960/41960 - 689s 16ms/step - loss: 0.1074 - acc: 0.9748 - val_loss: 0.0734 - val_acc: 0.9790
Epoch 2/10 - 41960/41960 - 650s 15ms/step - loss: 0.0657 - acc: 0.9809 - val_loss: 0.0539 - val_acc: 0.9826
Epoch 3/10 - 41960/41960 - 649s 15ms/step - loss: 0.0527 - acc: 0.9833 - val_loss: 0.0481 - val_acc: 0.9832
Epoch 4/10 - 41960/41960 - 649s 15ms/step - loss: 0.0388 - acc: 0.9865 - val_loss: 0.0474 - val_acc: 0.9818
Epoch 5/10 - 41960/41960 - 655s 16ms/step - loss: 0.0351 - acc: 0.9871 - val_loss: 0.0445 - val_acc: 0.9855

Epoch 6/10 - 41960/41960 - 799s 19ms/step - loss: 0.0328 - acc: 0.9882 - val_loss: 0.0478 - val_acc: 0.9861
Epoch 7/10 - 41960/41960 - 1149s 27ms/step - loss: 0.0306 - acc: 0.9887 - val_loss: 0.0475 - val_acc: 0.9854
Epoch 8/10 - 41960/41960 - 1145s 27ms/step - loss: 0.0290 - acc: 0.9890 - val_loss: 0.0423 - val_acc: 0.9862
Epoch 9/10 - 41960/41960 - 1473s 35ms/step - loss: 0.0283 - acc: 0.9897 - val_loss: 0.0423 - val_acc: 0.9845
Epoch 10/10 - 41960/41960 - 1792s 43ms/step - loss: 0.0269 - acc: 0.9905 - val_loss: 0.0425 - val_acc: 0.9856

**Dataset 2:**
Train on 21040 samples, validate on 5261 samples
Epoch 1/10 - 21040/21040 - 462s 22ms/step - loss: 0.6263 - acc: 0.6771 - val_loss: 0.6046 - val_acc: 0.7029
Epoch 2/10 - 21040/21040 - 453s 22ms/step - loss: 0.6081 - acc: 0.6914 - val_loss: 0.5925 - val_acc: 0.7075
Epoch 3/10 - 21040/21040 - 454s 22ms/step - loss: 0.5965 - acc: 0.7012 - val_loss: 0.5974 - val_acc: 0.7098
Epoch 4/10 - 21040/21040 - 383s 18ms/step - loss: 0.5913 - acc: 0.7035 - val_loss: 0.5994 - val_acc: 0.7079
Epoch 5/10 - 21040/21040 - 330s 16ms/step - loss: 0.5841 - acc: 0.7077 - val_loss: 0.6053 - val_acc: 0.7109
Epoch 6/10 - 21040/21040 - 330s 16ms/step - loss: 0.5765 - acc: 0.7119 - val_loss: 0.5977 - val_acc: 0.7025
Epoch 7/10 - 21040/21040 - 307s 15ms/step - loss: 0.5727 - acc: 0.7139 - val_loss: 0.6014 - val_acc: 0.7082
Epoch 8/10 - 21040/21040 - 233s 11ms/step - loss: 0.5688 - acc: 0.7154 - val_loss: 0.6087 - val_acc: 0.7079
Epoch 9/10 - 21040/21040 - 233s 11ms/step - loss: 0.5652 - acc: 0.7161 - val_loss: 0.6191 - val_acc: 0.6940
Epoch 10/10 - 21040/21040 - 233s 11ms/step - loss: 0.5643 - acc: 0.7153 - val_loss: 0.6107 - val_acc: 0.7109

**Dataset 3:**
Train on 58620 samples, validate on 14655 samples
Epoch 1/10 - 58620/58620 - 1003s 17ms/step - loss: 0.5433 - acc: 0.7510 - val_loss: 0.5236 - val_acc: 0.7639
Epoch 2/10 - 58620/58620 - 787s 13ms/step - loss: 0.5187 - acc: 0.7644 - val_loss: 0.5116 - val_acc: 0.7713
Epoch 3/10 - 58620/58620 - 1026s 18ms/step - loss: 0.5024 - acc: 0.7757 - val_loss: 0.5001 - val_acc: 0.7779
Epoch 4/10 - 58620/58620 - 664s 11ms/step - loss: 0.4944 - acc: 0.7795 - val_loss: 0.5001 - val_acc: 0.7764
Epoch 5/10 - 58620/58620 - 643s 11ms/step - loss: 0.4977 - acc: 0.7785 - val_loss: 0.5016 - val_acc: 0.7778
Epoch 6/10 - 58620/58620 - 648s 11ms/step - loss: 0.4893 - acc: 0.7834 - val_loss: 0.4953 - val_acc: 0.7784
Epoch 7/10 - 58620/58620 - 647s 11ms/step - loss: 0.4881 - acc: 0.7848 - val_loss: 0.4975 - val_acc: 0.7795

Epoch 8/10 - 58620/58620 - 642s 11ms/step - loss: 0.4848 - acc: 0.7861 - val_loss: 0.4982 - val_acc: 0.7796
Epoch 9/10 - 58620/58620 - 643s 11ms/step - loss: 0.4820 - acc: 0.7870 - val_loss: 0.4998 - val_acc: 0.7776
Epoch 10/10 - 58620/58620 - 643s 11ms/step - loss: 0.4790 - acc: 0.7889 - val_loss: 0.5009 - val_acc: 0.7784

*GRU + Dropout*

**Dataset 1:**
Train on 41960 samples, validate on 10490 samples
Epoch 1/10 - 41960/41960 - 520s 12ms/step - loss: 0.1136 - acc: 0.9722 - val_loss: 0.0681 - val_acc: 0.9796
Epoch 2/10 - 41960/41960 - 549s 13ms/step - loss: 0.0677 - acc: 0.9784 - val_loss: 0.1035 - val_acc: 0.9704
Epoch 3/10 - 41960/41960 - 554s 13ms/step - loss: 0.0575 - acc: 0.9809 - val_loss: 0.0490 - val_acc: 0.9829
Epoch 4/10 - 41960/41960 - 548s 13ms/step - loss: 0.0347 - acc: 0.9874 - val_loss: 0.0396 - val_acc: 0.9870
Epoch 5/10 - 41960/41960 - 548s 13ms/step - loss: 0.0304 - acc: 0.9885 - val_loss: 0.0437 - val_acc: 0.9871
Epoch 6/10 - 41960/41960 - 554s 13ms/step - loss: 0.0284 - acc: 0.9893 - val_loss: 0.0374 - val_acc: 0.9869
Epoch 7/10 - 41960/41960 - 841s 20ms/step - loss: 0.0274 - acc: 0.9894 - val_loss: 0.0360 - val_acc: 0.9883
Epoch 8/10 - 41960/41960 - 658s 16ms/step - loss: 0.0260 - acc: 0.9903 - val_loss: 0.0395 - val_acc: 0.9849
Epoch 9/10 - 41960/41960 - 832s 20ms/step - loss: 0.0251 - acc: 0.9902 - val_loss: 0.0402 - val_acc: 0.9862
Epoch 10/10 - 41960/41960 - 864s 21ms/step - loss: 0.0237 - acc: 0.9908 - val_loss: 0.0419 - val_acc: 0.9867

**Dataset 2:**
Train on 21040 samples, validate on 5261 samples
Epoch 1/15 - 21040/21040 - 192s 9ms/step - loss: 0.6266 - acc: 0.6807 - val_loss: 0.6170 - val_acc: 0.6852
Epoch 2/15 - 21040/21040 - 339s 16ms/step - loss: 0.6125 - acc: 0.6878 - val_loss: 0.5997 - val_acc: 0.7037
Epoch 3/15 - 21040/21040 - 340s 16ms/step - loss: 0.5970 - acc: 0.7035 - val_loss: 0.6045 - val_acc: 0.6945
Epoch 4/15 - 21040/21040 - 341s 16ms/step - loss: 0.5919 - acc: 0.7051 - val_loss: 0.5991 - val_acc: 0.7027
Epoch 5/15 - 21040/21040 - 339s 16ms/step - loss: 0.5862 - acc: 0.7074 - val_loss: 0.5969 - val_acc: 0.7044
Epoch 6/15 - 21040/21040 - 204s 10ms/step - loss: 0.5817 - acc: 0.7120 - val_loss: 0.5945 - val_acc: 0.7058
Epoch 7/15 - 21040/21040 - 160s 8ms/step - loss: 0.5779 - acc: 0.7136 - val_loss: 0.5983 - val_acc: 0.7033
Epoch 8/15 - 21040/21040 - 159s 8ms/step - loss: 0.5740 - acc: 0.7155 - val_loss: 0.6035 - val_acc: 0.7039
Epoch 9/15 - 21040/21040 - 154s 7ms/step - loss: 0.5716 - acc: 0.7146 - val_loss: 0.6089 - val_acc: 0.7050

Epoch 10/15 - 21040/21040 - 116s 5ms/step - loss: 0.5703 - acc: 0.7165 - val_loss: 0.6047 - val_acc: 0.7029
Epoch 11/15 - 21040/21040 - 116s 5ms/step - loss: 0.5651 - acc: 0.7179 - val_loss: 0.6089 - val_acc: 0.6995
Epoch 12/15 - 21040/21040 - 116s 5ms/step - loss: 0.5799 - acc: 0.7075 - val_loss: 0.6104 - val_acc: 0.7001
Epoch 13/15 - 21040/21040 - 116s 5ms/step - loss: 0.5622 - acc: 0.7184 - val_loss: 0.6128 - val_acc: 0.7001
Epoch 14/15 - 21040/21040 - 116s 5ms/step - loss: 0.5579 - acc: 0.7207 - val_loss: 0.6266 - val_acc: 0.6980
Epoch 15/15 - 21040/21040 - 116s 5ms/step - loss: 0.5574 - acc: 0.7228 - val_loss: 0.6206 - val_acc: 0.6985

**Dataset 3:**
Train on 2216 samples, validate on 554 samples
Epoch 1/10 - 2216/2216 - 54s 24ms/step - loss: 0.5697 - acc: 0.6832 - val_loss: 0.4561 - val_acc: 0.7762
Epoch 2/10 - 2216/2216 - 53s 24ms/step - loss: 0.4120 - acc: 0.8089 - val_loss: 0.3012 - val_acc: 0.8899
Epoch 3/10 - 2216/2216 - 49s 22ms/step - loss: 0.2985 - acc: 0.8883 - val_loss: 0.4100 - val_acc: 0.8159
Epoch 4/10 - 2216/2216 - 49s 22ms/step - loss: 0.2612 - acc: 0.9066 - val_loss: 0.2736 - val_acc: 0.9034
Epoch 5/10 - 2216/2216 - 48s 22ms/step - loss: 0.2112 - acc: 0.9285 - val_loss: 0.3041 - val_acc: 0.9007
Epoch 6/10 - 2216/2216 - 51s 23ms/step - loss: 0.2609 - acc: 0.9075 - val_loss: 0.2511 - val_acc: 0.9025
Epoch 7/10 - 2216/2216 - 55s 25ms/step - loss: 0.2031 - acc: 0.9296 - val_loss: 0.2510 - val_acc: 0.9061
Epoch 8/10 - 2216/2216 - 48s 22ms/step - loss: 0.1821 - acc: 0.9341 - val_loss: 0.3400 - val_acc: 0.8556
Epoch 9/10 - 2216/2216 - 37s 17ms/step - loss: 0.2255 - acc: 0.9264 - val_loss: 0.2284 - val_acc: 0.9224
Epoch 10/10 - 2216/2216 - 62s 28ms/step - loss: 0.1606 - acc: 0.9488 - val_loss: 0.1929 - val_acc: 0.9305

*LSTM Bidirectional*

**Dataset 1:**
Train on 41960 samples, validate on 10490 samples
Epoch 1/10 - 41960/41960 - 588s 14ms/step - loss: 0.0933 - acc: 0.9748 - val_loss: 0.0536 - val_acc: 0.9827
Epoch 2/10 - 41960/41960 - 553s 13ms/step - loss: 0.0495 - acc: 0.9826 - val_loss: 0.0473 - val_acc: 0.9844
Epoch 3/10 - 41960/41960 - 468s 11ms/step - loss: 0.0381 - acc: 0.9864 - val_loss: 0.0422 - val_acc: 0.9868
Epoch 4/10 - 41960/41960 - 426s 10ms/step - loss: 0.0335 - acc: 0.9879 - val_loss: 0.0412 - val_acc: 0.9874
Epoch 5/10 - 41960/41960 - 431s 10ms/step - loss: 0.0291 - acc: 0.9897 - val_loss: 0.0415 - val_acc: 0.9871
Epoch 6/10 - 41960/41960 - 538s 13ms/step - loss: 0.0260 - acc: 0.9905 - val_loss: 0.0444 - val_acc: 0.9854

Epoch 7/10 - 41960/41960 - 610s 15ms/step - loss: 0.0247 - acc: 0.9914 - val_loss: 0.0446 - val_acc: 0.9861
Epoch 8/10 - 41960/41960 - 494s 12ms/step - loss: 0.0228 - acc: 0.9921 - val_loss: 0.0561 - val_acc: 0.9850
Epoch 9/10 - 41960/41960 - 424s 10ms/step - loss: 0.0223 - acc: 0.9921 - val_loss: 0.0451 - val_acc: 0.9852
Epoch 10/10 - 41960/41960 - 415s 10ms/step - loss: 0.0176 - acc: 0.9938 - val_loss: 0.0469 - val_acc: 0.9870

**Dataset 2:**
Train on 21040 samples, validate on 5261 samples
Epoch 1/8 - 21040/21040 - 169s 8ms/step - loss: 0.6205 - acc: 0.6838 - val_loss: 0.6113 - val_acc: 0.6976
Epoch 2/8 - 21040/21040 - 189s 9ms/step - loss: 0.5986 - acc: 0.7008 - val_loss: 0.6031 - val_acc: 0.6949
Epoch 3/8 - 21040/21040 - 183s 9ms/step - loss: 0.5913 - acc: 0.7048 - val_loss: 0.6027 - val_acc: 0.7001
Epoch 4/8 - 21040/21040 - 184s 9ms/step - loss: 0.5814 - acc: 0.7102 - val_loss: 0.6070 - val_acc: 0.6999
Epoch 5/8 - 21040/21040 - 183s 9ms/step - loss: 0.5778 - acc: 0.7122 - val_loss: 0.6074 - val_acc: 0.6909
Epoch 6/8 - 21040/21040 - 184s 9ms/step - loss: 0.5714 - acc: 0.7154 - val_loss: 0.6041 - val_acc: 0.6999
Epoch 7/8 - 21040/21040 - 183s 9ms/step - loss: 0.5654 - acc: 0.7188 - val_loss: 0.6080 - val_acc: 0.6942
Epoch 8/8 - 21040/21040 - 183s 9ms/step - loss: 0.5609 - acc: 0.7216 - val_loss: 0.6133 - val_acc: 0.6911

**Dataset 3:**
Train on 58620 samples, validate on 14655 samples
Epoch 1/8 - 58620/58620 - 626s 11ms/step - loss: 0.5414 - acc: 0.7504 - val_loss: 0.5253 - val_acc: 0.7545
Epoch 2/8 - 58620/58620 - 511s 9ms/step - loss: 0.5077 - acc: 0.7702 - val_loss: 0.5055 - val_acc: 0.7702
Epoch 3/8 - 58620/58620 - 502s 9ms/step - loss: 0.4954 - acc: 0.7806 - val_loss: 0.5024 - val_acc: 0.7726
Epoch 4/8 - 58620/58620 - 505s 9ms/step - loss: 0.4903 - acc: 0.7824 - val_loss: 0.5011 - val_acc: 0.7745
Epoch 5/8 - 58620/58620 - 502s 9ms/step - loss: 0.4856 - acc: 0.7849 - val_loss: 0.4998 - val_acc: 0.7773
Epoch 6/8 - 58620/58620 - 518s 9ms/step - loss: 0.4817 - acc: 0.7874 - val_loss: 0.4979 - val_acc: 0.7783
Epoch 7/8 - 58620/58620 - 528s 9ms/step - loss: 0.4772 - acc: 0.7899 - val_loss: 0.5045 - val_acc: 0.7745
Epoch 8/8 - 58620/58620 - 571s 10ms/step - loss: 0.4730 - acc: 0.7915 - val_loss: 0.5029 - val_acc: 0.7752

*LSTM Bidirectional + Dropout*

**Dataset 1:**

Train on 41960 samples, validate on 10490 samples
Epoch 1/10 - 41960/41960 - 564s 13ms/step - loss: 0.1086 - acc: 0.9737 - val_loss: 0.0634 - val_acc: 0.9724
Epoch 2/10 - 41960/41960 - 561s 13ms/step - loss: 0.0646 - acc: 0.9786 - val_loss: 0.0525 - val_acc: 0.9817
Epoch 3/10 - 41960/41960 - 476s 11ms/step - loss: 0.0472 - acc: 0.9847 - val_loss: 0.0492 - val_acc: 0.9843
Epoch 4/10 - 41960/41960 - 432s 10ms/step - loss: 0.0419 - acc: 0.9864 - val_loss: 0.0396 - val_acc: 0.9863
Epoch 5/10 - 41960/41960 - 438s 10ms/step - loss: 0.0356 - acc: 0.9884 - val_loss: 0.0387 - val_acc: 0.9875
Epoch 6/10 - 41960/41960 - 547s 13ms/step - loss: 0.0328 - acc: 0.9893 - val_loss: 0.0382 - val_acc: 0.9878
Epoch 7/10 - 41960/41960 - 619s 15ms/step - loss: 0.0300 - acc: 0.9900 - val_loss: 0.0457 - val_acc: 0.9848
Epoch 8/10 - 41960/41960 - 497s 12ms/step - loss: 0.0272 - acc: 0.9908 - val_loss: 0.0443 - val_acc: 0.9858
Epoch 9/10 - 41960/41960 - 428s 10ms/step - loss: 0.0249 - acc: 0.9919 - val_loss: 0.0410 - val_acc: 0.9875
Epoch 10/10 - 41960/41960 - 409s 10ms/step - loss: 0.0232 - acc: 0.9923 - val_loss: 0.0468 - val_acc: 0.9859

**Dataset 2:**
Train on 21040 samples, validate on 5261 samples
Epoch 1/8 - 21040/21040 - 199s 9ms/step - loss: 0.6200 - acc: 0.6868 - val_loss: 0.6185 - val_acc: 0.6807
Epoch 2/8 - 21040/21040 - 181s 9ms/step - loss: 0.5990 - acc: 0.7042 - val_loss: 0.6113 - val_acc: 0.6907
Epoch 3/8 - 21040/21040 - 181s 9ms/step - loss: 0.5896 - acc: 0.7075 - val_loss: 0.6107 - val_acc: 0.6892
Epoch 4/8 - 21040/21040 - 180s 9ms/step - loss: 0.5853 - acc: 0.7103 - val_loss: 0.6149 - val_acc: 0.6881
Epoch 5/8 - 21040/21040 - 180s 9ms/step - loss: 0.5782 - acc: 0.7170 - val_loss: 0.6126 - val_acc: 0.6883
Epoch 6/8 - 21040/21040 - 181s 9ms/step - loss: 0.5707 - acc: 0.7222 - val_loss: 0.6204 - val_acc: 0.6921
Epoch 7/8 - 21040/21040 - 180s 9ms/step - loss: 0.5690 - acc: 0.7226 - val_loss: 0.6192 - val_acc: 0.6921
Epoch 8/8 - 21040/21040 - 156s 7ms/step - loss: 0.5664 - acc: 0.7234 - val_loss: 0.6233 - val_acc: 0.6871

**Dataset 3:**
Train on 58620 samples, validate on 14655 samples
Epoch 1/8 - 58620/58620 - 615s 10ms/step - loss: 0.5424 - acc: 0.7508 - val_loss: 0.5260 - val_acc: 0.7611
Epoch 2/8 - 58620/58620 - 504s 9ms/step - loss: 0.5078 - acc: 0.7743 - val_loss: 0.5148 - val_acc: 0.7687
Epoch 3/8 - 58620/58620 - 503s 9ms/step - loss: 0.4994 - acc: 0.7773 - val_loss: 0.5106 - val_acc: 0.7677
Epoch 4/8 - 58620/58620 - 506s 9ms/step - loss: 0.4931 - acc: 0.7823 - val_loss: 0.5071 - val_acc: 0.7702

Epoch 5/8 - 58620/58620 - 504s 9ms/step - loss: 0.4900 - acc: 0.7845 - val_loss: 0.5073 - val_acc: 0.7724
Epoch 6/8 - 58620/58620 - 521s 9ms/step - loss: 0.4864 - acc: 0.7860 - val_loss: 0.5105 - val_acc: 0.7679
Epoch 7/8 - 58620/58620 - 553s 9ms/step - loss: 0.4810 - acc: 0.7885 - val_loss: 0.5073 - val_acc: 0.7725
Epoch 8/8 - 58620/58620 - 515s 9ms/step - loss: 0.4768 - acc: 0.7909 - val_loss: 0.5162 - val_acc: 0.7717

## Appendix B Training Results for Pre-Trained Word Embedding Models

*CNN*

**Dataset 1:**
Train on 2216 samples, validate on 554 samples
Epoch 1/10 - 2216/2216 - 65s 29ms/step - loss: 0.4927 - acc: 0.7739 - val_loss: 0.2307 - val_acc: 0.9260
Epoch 2/10 - 2216/2216 - 62s 28ms/step - loss: 0.1647 - acc: 0.9393 - val_loss: 0.1219 - val_acc: 0.9477
Epoch 3/10 - 2216/2216 - 62s 28ms/step - loss: 0.1018 - acc: 0.9594 - val_loss: 0.1109 - val_acc: 0.9504
Epoch 4/10 - 2216/2216 - 61s 28ms/step - loss: 0.0739 - acc: 0.9700 - val_loss: 0.0919 - val_acc: 0.9693
Epoch 5/10 - 2216/2216 - 62s 28ms/step - loss: 0.0499 - acc: 0.9871 - val_loss: 0.0965 - val_acc: 0.9684
Epoch 6/10 - 2216/2216 - 62s 28ms/step - loss: 0.0325 - acc: 0.9941 - val_loss: 0.0960 - val_acc: 0.9666
Epoch 7/10 - 2216/2216 - 62s 28ms/step - loss: 0.0200 - acc: 0.9977 - val_loss: 0.0910 - val_acc: 0.9693
Epoch 8/10 - 2216/2216 - 62s 28ms/step - loss: 0.0118 - acc: 0.9993 - val_loss: 0.0951 - val_acc: 0.9684
Epoch 9/10 - 2216/2216 - 62s 28ms/step - loss: 0.0079 - acc: 1.0000 - val_loss: 0.0908 - val_acc: 0.9711
Epoch 10/10 - 2216/2216 - 57s 26ms/step - loss: 0.0053 - acc: 1.0000 - val_loss: 0.1085 - val_acc: 0.9657

**Dataset 2:**
Train on 979 samples, validate on 244 samples
Epoch 1/10 - 979/979 - 13s 13ms/step - loss: 0.6684 - acc: 0.6093 - val_loss: 0.6289 - val_acc: 0.6680
Epoch 2/10 - 979/979 - 12s 12ms/step - loss: 0.6061 - acc: 0.6747 - val_loss: 0.6530 - val_acc: 0.6209
Epoch 3/10 - 979/979 - 12s 12ms/step - loss: 0.5667 - acc: 0.7140 - val_loss: 0.5792 - val_acc: 0.6988
Epoch 4/10 - 979/979 - 12s 13ms/step - loss: 0.5244 - acc: 0.7375 - val_loss: 0.6238 - val_acc: 0.6455
Epoch 5/10 - 979/979 - 13s 13ms/step - loss: 0.4800 - acc: 0.7840 - val_loss: 0.5737 - val_acc: 0.7234
Epoch 6/10 - 979/979 - 13s 13ms/step - loss: 0.4525 - acc: 0.7932 - val_loss: 0.6589 - val_acc: 0.6352
Epoch 7/10 - 979/979 - 14s 14ms/step - loss: 0.3798 - acc: 0.8493 - val_loss: 0.6105 - val_acc: 0.6906

Epoch 8/10 - 979/979 - 12s 13ms/step - loss: 0.3265 - acc: 0.8810 - val_loss: 0.6067 - val_acc: 0.7029
Epoch 9/10 - 979/979 - 12s 13ms/step - loss: 0.2692 - acc: 0.9162 - val_loss: 0.6210 - val_acc: 0.6988
Epoch 10/10 - 979/979 - 12s 12ms/step - loss: 0.2290 - acc: 0.9346 - val_loss: 0.8152 - val_acc: 0.6148

**Dataset 3:**
Train on 2981 samples, validate on 745 samples
Epoch 1/10 - 2981/2981 - 51s 17ms/step - loss: 0.5233 - acc: 0.7444 - val_loss: 0.3804 - val_acc: 0.8114
Epoch 2/10 - 2981/2981 - 54s 18ms/step - loss: 0.3178 - acc: 0.8564 - val_loss: 0.3194 - val_acc: 0.8483
Epoch 3/10 - 2981/2981 - 61s 20ms/step - loss: 0.2480 - acc: 0.8900 - val_loss: 0.3005 - val_acc: 0.8577
Epoch 4/10 - 2981/2981 - 82s 28ms/step - loss: 0.2205 - acc: 0.9031 - val_loss: 0.3252 - val_acc: 0.8597
Epoch 5/10 - 2981/2981 - 83s 28ms/step - loss: 0.1873 - acc: 0.9242 - val_loss: 0.2962 - val_acc: 0.8758
Epoch 6/10 - 2981/2981 - 83s 28ms/step - loss: 0.1365 - acc: 0.9520 - val_loss: 0.3189 - val_acc: 0.8685
Epoch 7/10 - 2981/2981 - 84s 28ms/step - loss: 0.1052 - acc: 0.9668 - val_loss: 0.3061 - val_acc: 0.8758
Epoch 8/10 - 2981/2981 - 83s 28ms/step - loss: 0.0656 - acc: 0.9866 - val_loss: 0.3218 - val_acc: 0.8805
Epoch 9/10 - 2981/2981 - 82s 28ms/step - loss: 0.0417 - acc: 0.9928 - val_loss: 0.3498 - val_acc: 0.8785
Epoch 10/10 - 2981/2981 - 85s 29ms/step - loss: 0.0246 - acc: 0.9973 - val_loss: 0.3523 - val_acc: 0.8799

*RCNN*

**Dataset 1:**
Train on 2216 samples, validate on 554 samples
Epoch 1/10 - 2216/2216 - 44s 20ms/step - loss: 0.4142 - acc: 0.7954 - val_loss: 0.1333 - val_acc: 0.9567
Epoch 2/10 - 2216/2216 - 40s 18ms/step - loss: 0.1230 - acc: 0.9562 - val_loss: 0.1138 - val_acc: 0.9567
Epoch 3/10 - 2216/2216 - 41s 19ms/step - loss: 0.1001 - acc: 0.9610 - val_loss: 0.1023 - val_acc: 0.9567
Epoch 4/10 - 2216/2216 - 48s 21ms/step - loss: 0.0784 - acc: 0.9659 - val_loss: 0.1263 - val_acc: 0.9341
Epoch 5/10 - 2216/2216 - 48s 22ms/step - loss: 0.0851 - acc: 0.9639 - val_loss: 0.1072 - val_acc: 0.9540
Epoch 6/10 - 2216/2216 - 61s 28ms/step - loss: 0.0753 - acc: 0.9713 - val_loss: 0.1788 - val_acc: 0.9431
Epoch 7/10 - 2216/2216 - 61s 28ms/step - loss: 0.0775 - acc: 0.9725 - val_loss: 0.1095 - val_acc: 0.9702
Epoch 8/10 - 2216/2216 - 62s 28ms/step - loss: 0.0436 - acc: 0.9878 - val_loss: 0.1004 - val_acc: 0.9711
Epoch 9/10 - 2216/2216 - 56s 25ms/step - loss: 0.0316 - acc: 0.9926 - val_loss: 0.1233 - val_acc: 0.9720

Epoch 10/10 - 2216/2216 - 56s 25ms/step - loss: 0.0340 - acc: 0.9919 - val_loss: 0.1118 - val_acc: 0.9702

**Dataset 2:**
Train on 979 samples, validate on 244 samples
Epoch 1/10 - 979/979 - 27s 27ms/step - loss: 0.6733 - acc: 0.6001 - val_loss: 0.6648 - val_acc: 0.5922
Epoch 2/10 - 979/979 - 23s 24ms/step - loss: 0.6402 - acc: 0.6195 - val_loss: 0.6522 - val_acc: 0.6045
Epoch 3/10 - 979/979 - 23s 24ms/step - loss: 0.5982 - acc: 0.7033 - val_loss: 0.6792 - val_acc: 0.6516
Epoch 4/10 - 979/979 - 23s 23ms/step - loss: 0.5872 - acc: 0.6997 - val_loss: 0.6735 - val_acc: 0.6373
Epoch 5/10 - 979/979 - 23s 24ms/step - loss: 0.5644 - acc: 0.7114 - val_loss: 0.6341 - val_acc: 0.6250
Epoch 6/10 - 979/979 - 23s 24ms/step - loss: 0.5337 - acc: 0.7451 - val_loss: 0.6544 - val_acc: 0.6352
Epoch 7/10 - 979/979 - 23s 24ms/step - loss: 0.5164 - acc: 0.7697 - val_loss: 0.6341 - val_acc: 0.6496
Epoch 8/10 - 979/979 - 23s 23ms/step - loss: 0.5061 - acc: 0.7732 - val_loss: 0.6388 - val_acc: 0.6270
Epoch 9/10 - 979/979 - 23s 23ms/step - loss: 0.4645 - acc: 0.8166 - val_loss: 0.7012 - val_acc: 0.6496
Epoch 10/10 - 979/979 - 23s 24ms/step - loss: 0.4447 - acc: 0.8100 - val_loss: 0.6725 - val_acc: 0.6332

**Dataset 3:**
Train on 2981 samples, validate on 745 samples
Epoch 1/10 - 2981/2981 - 67s 23ms/step - loss: 0.5374 - acc: 0.7196 - val_loss: 0.4120 - val_acc: 0.7899
Epoch 2/10 - 2981/2981 - 57s 19ms/step - loss: 0.3408 - acc: 0.8375 - val_loss: 0.3565 - val_acc: 0.8195
Epoch 3/10 - 2981/2981 - 56s 19ms/step - loss: 0.2922 - acc: 0.8640 - val_loss: 0.3578 - val_acc: 0.8255
Epoch 4/10 - 2981/2981 - 54s 18ms/step - loss: 0.2829 - acc: 0.8678 - val_loss: 0.3448 - val_acc: 0.8389
Epoch 5/10 - 2981/2981 - 55s 19ms/step - loss: 0.2373 - acc: 0.8940 - val_loss: 0.3464 - val_acc: 0.8436
Epoch 6/10 - 2981/2981 - 58s 19ms/step - loss: 0.2387 - acc: 0.8977 - val_loss: 0.3490 - val_acc: 0.8275
Epoch 7/10 - 2981/2981 - 58s 19ms/step - loss: 0.2080 - acc: 0.9136 - val_loss: 0.3503 - val_acc: 0.8376
Epoch 8/10 - 2981/2981 - 59s 20ms/step - loss: 0.1716 - acc: 0.9301 - val_loss: 0.3691 - val_acc: 0.8456
Epoch 9/10 - 2981/2981 - 58s 19ms/step - loss: 0.1348 - acc: 0.9550 - val_loss: 0.4188 - val_acc: 0.8228
Epoch 10/10 - 2981/2981 - 68s 23ms/step - loss: 0.1230 - acc: 0.9577 - val_loss: 0.4382 - val_acc: 0.8362

*LSTM*

**Dataset 1:**

Train on 2216 samples, validate on 554 samples
Epoch 1/10 - 2216/2216 - 240s 108ms/step - loss: 0.5546 - acc: 0.6997 - val_loss: 0.4669 - val_acc: 0.7518
Epoch 2/10 - 2216/2216 - 108s 49ms/step - loss: 0.3849 - acc: 0.8139 - val_loss: 0.2705 - val_acc: 0.8980
Epoch 3/10 - 2216/2216 - 99s 45ms/step - loss: 0.3434 - acc: 0.8606 - val_loss: 0.3644 - val_acc: 0.8556
Epoch 4/10 - 2216/2216 - 130s 59ms/step - loss: 0.2131 - acc: 0.9255 - val_loss: 0.2322 - val_acc: 0.9161
Epoch 5/10 - 2216/2216 - 51s 23ms/step - loss: 0.1794 - acc: 0.9341 - val_loss: 0.2056 - val_acc: 0.9215
Epoch 6/10 - 2216/2216 - 50s 23ms/step - loss: 0.1516 - acc: 0.9470 - val_loss: 0.2241 - val_acc: 0.9116
Epoch 7/10 - 2216/2216 - 52s 24ms/step - loss: 0.1393 - acc: 0.9574 - val_loss: 0.2045 - val_acc: 0.9233
Epoch 8/10 - 2216/2216 - 55s 25ms/step - loss: 0.1309 - acc: 0.9594 - val_loss: 0.1746 - val_acc: 0.9359
Epoch 9/10 - 2216/2216 - 54s 24ms/step - loss: 0.1262 - acc: 0.9571 - val_loss: 0.1968 - val_acc: 0.9260
Epoch 10/10 - 2216/2216 - 56s 25ms/step - loss: 0.1188 - acc: 0.9576 - val_loss: 0.1504 - val_acc: 0.9386

**Dataset 2:**
Train on 979 samples, validate on 244 samples
Epoch 1/10 - 979/979 - 28s 28ms/step - loss: 0.6708 - acc: 0.5756 - val_loss: 0.6438 - val_acc: 0.6189
Epoch 2/10 - 979/979 - 25s 26ms/step - loss: 0.6683 - acc: 0.6522 - val_loss: 0.6084 - val_acc: 0.7336
Epoch 3/10 - 979/979 - 25s 25ms/step - loss: 0.6368 - acc: 0.6333 - val_loss: 0.6436 - val_acc: 0.6783
Epoch 4/10 - 979/979 - 25s 25ms/step - loss: 0.6404 - acc: 0.6629 - val_loss: 0.6491 - val_acc: 0.6783
Epoch 5/10 - 979/979 - 25s 25ms/step - loss: 0.6298 - acc: 0.6619 - val_loss: 0.6344 - val_acc: 0.6701
Epoch 6/10 - 979/979 - 25s 25ms/step - loss: 0.6122 - acc: 0.6828 - val_loss: 0.6067 - val_acc: 0.6742
Epoch 7/10 - 979/979 - 25s 25ms/step - loss: 0.5994 - acc: 0.7099 - val_loss: 0.6451 - val_acc: 0.6516
Epoch 8/10 - 979/979 - 25s 25ms/step - loss: 0.5950 - acc: 0.7068 - val_loss: 0.6254 - val_acc: 0.6619
Epoch 9/10 - 979/979 - 25s 25ms/step - loss: 0.5718 - acc: 0.7211 - val_loss: 0.6118 - val_acc: 0.6762
Epoch 10/10 - 979/979 - 25s 25ms/step - loss: 0.5635 - acc: 0.7298 - val_loss: 0.6208 - val_acc: 0.6844

**Dataset 3:**
Train on 58620 samples, validate on 14655 samples
Epoch 1/10 - 58620/58620 - 1034s 18ms/step - loss: 0.5456 - acc: 0.7490 - val_loss: 0.5321 - val_acc: 0.7629
Epoch 2/10 - 58620/58620 - 931s 16ms/step - loss: 0.5129 - acc: 0.7682 - val_loss: 0.5186 - val_acc: 0.7662

Epoch 3/10 - 58620/58620 - 929s 16ms/step - loss: 0.5021 - acc: 0.7764 - val_loss: 0.5132 - val_acc: 0.7727
Epoch 4/10 - 58620/58620 - 982s 17ms/step - loss: 0.4961 - acc: 0.7796 - val_loss: 0.5060 - val_acc: 0.7735
Epoch 5/10 - 58620/58620 - 805s 14ms/step - loss: 0.4930 - acc: 0.7810 - val_loss: 0.5058 - val_acc: 0.7756
Epoch 6/10 - 58620/58620 - 868s 15ms/step - loss: 0.4893 - acc: 0.7833 - val_loss: 0.5078 - val_acc: 0.7754
Epoch 7/10 - 58620/58620 - 715s 12ms/step - loss: 0.4872 - acc: 0.7845 - val_loss: 0.5060 - val_acc: 0.7778
Epoch 8/10 - 58620/58620 - 437s 7ms/step - loss: 0.4841 - acc: 0.7869 - val_loss: 0.5037 - val_acc: 0.7766
Epoch 9/10 - 58620/58620 - 558s 10ms/step - loss: 0.4813 - acc: 0.7882 - val_loss: 0.5053 - val_acc: 0.7758
Epoch 10/10 - 58620/58620 - 561s 10ms/step - loss: 0.4783 - acc: 0.7899 - val_loss: 0.5086 - val_acc: 0.7756

*Double LSTM*

**Dataset 1:**
Train on 2216 samples, validate on 554 samples
Epoch 1/10 - 2216/2216 - 108s 49ms/step - loss: 0.5414 - acc: 0.7067 - val_loss: 0.3529 - val_acc: 0.8736
Epoch 2/10 - 2216/2216 - 106s 48ms/step - loss: 0.2605 - acc: 0.9106 - val_loss: 0.1956 - val_acc: 0.9422
Epoch 3/10 - 2216/2216 - 110s 50ms/step - loss: 0.1995 - acc: 0.9343 - val_loss: 0.2188 - val_acc: 0.9278
Epoch 4/10 - 2216/2216 - 119s 54ms/step - loss: 0.2369 - acc: 0.9147 - val_loss: 0.2012 - val_acc: 0.9350
Epoch 5/10 - 2216/2216 - 119s 54ms/step - loss: 0.1578 - acc: 0.9499 - val_loss: 0.1797 - val_acc: 0.9440
Epoch 6/10 - 2216/2216 - 108s 49ms/step - loss: 0.1246 - acc: 0.9596 - val_loss: 0.1763 - val_acc: 0.9468
Epoch 7/10 - 2216/2216 - 109s 49ms/step - loss: 0.1495 - acc: 0.9495 - val_loss: 0.2221 - val_acc: 0.9314
Epoch 8/10 - 2216/2216 - 104s 47ms/step - loss: 0.1218 - acc: 0.9587 - val_loss: 0.1064 - val_acc: 0.9639
Epoch 9/10 - 2216/2216 - 105s 47ms/step - loss: 0.0889 - acc: 0.9707 - val_loss: 0.1082 - val_acc: 0.9639
Epoch 10/10 - 2216/2216 - 122s 55ms/step - loss: 0.0886 - acc: 0.9713 - val_loss: 0.1412 - val_acc: 0.9504

**Dataset 2:**
Train on 979 samples, validate on 244 samples
Epoch 1/10 - 979/979 - 82s 83ms/step - loss: 0.6699 - acc: 0.5822 - val_loss: 0.6461 - val_acc: 0.5943
Epoch 2/10 - 979/979 - 60s 61ms/step - loss: 0.6400 - acc: 0.6313 - val_loss: 0.6344 - val_acc: 0.6619
Epoch 3/10 - 979/979 - 58s 59ms/step - loss: 0.6239 - acc: 0.6604 - val_loss: 0.6034 - val_acc: 0.7111
Epoch 4/10 - 979/979 - 39s 39ms/step - loss: 0.5896 - acc: 0.6936 - val_loss: 0.6040 - val_acc: 0.6947

Epoch 5/10 - 979/979 - 29s 29ms/step - loss: 0.5873 - acc: 0.7074 - val_loss: 0.6190 - val_acc: 0.6885
Epoch 6/10 - 979/979 - 29s 29ms/step - loss: 0.5766 - acc: 0.7308 - val_loss: 0.6158 - val_acc: 0.6803
Epoch 7/10 - 979/979 - 28s 28ms/step - loss: 0.5567 - acc: 0.7436 - val_loss: 0.5940 - val_acc: 0.6803
Epoch 8/10 - 979/979 - 28s 28ms/step - loss: 0.5198 - acc: 0.7564 - val_loss: 0.5959 - val_acc: 0.6742
Epoch 9/10 - 979/979 - 28s 28ms/step - loss: 0.4839 - acc: 0.7988 - val_loss: 0.6467 - val_acc: 0.6762
Epoch 10/10 - 979/979 - 28s 29ms/step - loss: 0.5070 - acc: 0.7656 - val_loss: 0.6006 - val_acc: 0.6906

**Dataset 3:**
Train on 2594 samples, validate on 648 samples
Epoch 1/10 - 2594/2594 - 88s 34ms/step - loss: 0.5977 - acc: 0.6654 - val_loss: 0.4900 - val_acc: 0.7593
Epoch 2/10 - 2594/2594 - 86s 33ms/step - loss: 0.4497 - acc: 0.7868 - val_loss: 0.4033 - val_acc: 0.7901
Epoch 3/10 - 2594/2594 - 86s 33ms/step - loss: 0.4423 - acc: 0.7733 - val_loss: 0.3861 - val_acc: 0.8110
Epoch 4/10 - 2594/2594 - 86s 33ms/step - loss: 0.3888 - acc: 0.8071 - val_loss: 0.3649 - val_acc: 0.8272
Epoch 5/10 - 2594/2594 - 86s 33ms/step - loss: 0.4053 - acc: 0.8045 - val_loss: 0.3917 - val_acc: 0.8287
Epoch 6/10 - 2594/2594 - 83s 32ms/step - loss: 0.3604 - acc: 0.8263 - val_loss: 0.3511 - val_acc: 0.8434
Epoch 7/10 - 2594/2594 - 79s 30ms/step - loss: 0.3428 - acc: 0.8387 - val_loss: 0.4478 - val_acc: 0.7647
Epoch 8/10 - 2594/2594 - 76s 29ms/step - loss: 0.3817 - acc: 0.8279 - val_loss: 0.3593 - val_acc: 0.8372
Epoch 9/10 - 2594/2594 - 76s 29ms/step - loss: 0.3285 - acc: 0.8504 - val_loss: 0.4564 - val_acc: 0.7870
Epoch 10/10 - 2594/2594 - 75s 29ms/step - loss: 0.3968 - acc: 0.8196 - val_loss: 0.4100 - val_acc: 0.8079

*LSTM + Dropout*

**Dataset 1:**
Train on 2216 samples, validate on 554 samples
Epoch 1/10 - 2216/2216 - 56s 25ms/step - loss: 0.5798 - acc: 0.6656 - val_loss: 0.4566 - val_acc: 0.7690
Epoch 2/10 - 2216/2216 - 54s 24ms/step - loss: 0.4213 - acc: 0.7902 - val_loss: 0.4247 - val_acc: 0.7879
Epoch 3/10 - 2216/2216 - 53s 24ms/step - loss: 0.2891 - acc: 0.8777 - val_loss: 0.3307 - val_acc: 0.8899
Epoch 4/10 - 2216/2216 - 53s 24ms/step - loss: 0.3475 - acc: 0.8667 - val_loss: 0.2622 - val_acc: 0.9269
Epoch 5/10 - 2216/2216 - 53s 24ms/step - loss: 0.1965 - acc: 0.9357 - val_loss: 0.1783 - val_acc: 0.9350
Epoch 6/10 - 2216/2216 - 53s 24ms/step - loss: 0.1828 - acc: 0.9350 - val_loss: 0.2243 - val_acc: 0.9242

Epoch 7/10 - 2216/2216 - 53s 24ms/step - loss: 0.1884 - acc: 0.9339 - val_loss: 0.1640 - val_acc: 0.9531
Epoch 8/10 - 2216/2216 - 54s 24ms/step - loss: 0.1463 - acc: 0.9540 - val_loss: 0.1561 - val_acc: 0.9531
Epoch 9/10 - 2216/2216 - 53s 24ms/step - loss: 0.1281 - acc: 0.9569 - val_loss: 0.1311 - val_acc: 0.9540
Epoch 10/10 - 2216/2216 - 53s 24ms/step - loss: 0.4493 - acc: 0.8421 - val_loss: 0.4513 - val_acc: 0.7572

**Dataset 2:**
Train on 979 samples, validate on 244 samples
Epoch 1/10 - 979/979 - 28s 28ms/step - loss: 0.6756 - acc: 0.5802 - val_loss: 0.6729 - val_acc: 0.5861
Epoch 2/10 - 979/979 - 25s 26ms/step - loss: 0.6517 - acc: 0.5981 - val_loss: 0.6595 - val_acc: 0.6107
Epoch 3/10 - 979/979 - 26s 27ms/step - loss: 0.6284 - acc: 0.6573 - val_loss: 0.6285 - val_acc: 0.6639
Epoch 4/10 - 979/979 - 25s 25ms/step - loss: 0.6088 - acc: 0.6813 - val_loss: 0.6118 - val_acc: 0.6926
Epoch 5/10 - 979/979 - 25s 25ms/step - loss: 0.5918 - acc: 0.7033 - val_loss: 0.6270 - val_acc: 0.6906
Epoch 6/10 - 979/979 - 25s 25ms/step - loss: 0.5877 - acc: 0.6900 - val_loss: 0.6431 - val_acc: 0.6742
Epoch 7/10 - 979/979 - 25s 25ms/step - loss: 0.5947 - acc: 0.7120 - val_loss: 0.6337 - val_acc: 0.6906
Epoch 8/10 - 979/979 - 25s 26ms/step - loss: 0.5630 - acc: 0.7344 - val_loss: 0.5890 - val_acc: 0.6967
Epoch 9/10 - 979/979 - 25s 25ms/step - loss: 0.5486 - acc: 0.7400 - val_loss: 0.5913 - val_acc: 0.7008
Epoch 10/10 - 979/979 - 25s 25ms/step - loss: 0.5495 - acc: 0.7406 - val_loss: 0.6056 - val_acc: 0.6906

**Dataset 3:**
Train on 2594 samples, validate on 648 samples
Epoch 1/10 - 2594/2594 - 49s 19ms/step - loss: 0.5809 - acc: 0.6795 - val_loss: 0.4591 - val_acc: 0.7639
Epoch 2/10 - 2594/2594 - 49s 19ms/step - loss: 0.4518 - acc: 0.7733 - val_loss: 0.4096 - val_acc: 0.7832
Epoch 3/10 - 2594/2594 - 47s 18ms/step - loss: 0.4722 - acc: 0.7546 - val_loss: 0.4465 - val_acc: 0.7677
Epoch 4/10 - 2594/2594 - 48s 19ms/step - loss: 0.4087 - acc: 0.8045 - val_loss: 0.3962 - val_acc: 0.8063
Epoch 5/10 - 2594/2594 - 69s 27ms/step - loss: 0.4131 - acc: 0.8059 - val_loss: 0.3795 - val_acc: 0.8094
Epoch 6/10 - 2594/2594 - 105s 40ms/step - loss: 0.3821 - acc: 0.8169 - val_loss: 0.3946 - val_acc: 0.8017
Epoch 7/10 - 2594/2594 - 135s 52ms/step - loss: 0.3621 - acc: 0.8304 - val_loss: 0.3920 - val_acc: 0.8171
Epoch 8/10 - 2594/2594 - 165s 63ms/step - loss: 0.3526 - acc: 0.8423 - val_loss: 0.3767 - val_acc: 0.8110

Epoch 9/10 - 2594/2594 - 156s 60ms/step - loss: 0.3416 - acc: 0.8464 - val_loss: 0.4069 - val_acc: 0.8117
Epoch 10/10 - 2594/2594 - 155s 60ms/step - loss: 0.3626 - acc: 0.8263 - val_loss: 0.3882 - val_acc: 0.8025

*GRU*

**Dataset 1:**
Train on 2216 samples, validate on 554 samples
Epoch 1/10 - 2216/2216 - 60s 27ms/step - loss: 0.5757 - acc: 0.6611 - val_loss: 0.4653 - val_acc: 0.7735
Epoch 2/10 - 2216/2216 - 60s 27ms/step - loss: 0.4307 - acc: 0.7868 - val_loss: 0.3815 - val_acc: 0.8384
Epoch 3/10 - 2216/2216 - 70s 32ms/step - loss: 0.4546 - acc: 0.8166 - val_loss: 0.4102 - val_acc: 0.8014
Epoch 4/10 - 2216/2216 - 68s 31ms/step - loss: 0.4000 - acc: 0.8116 - val_loss: 0.3820 - val_acc: 0.8159
Epoch 5/10 - 2216/2216 - 67s 30ms/step - loss: 0.3139 - acc: 0.8601 - val_loss: 0.2391 - val_acc: 0.9233
Epoch 6/10 - 2216/2216 - 65s 29ms/step - loss: 0.4637 - acc: 0.8060 - val_loss: 0.4084 - val_acc: 0.7969
Epoch 7/10 - 2216/2216 - 66s 30ms/step - loss: 0.3021 - acc: 0.8770 - val_loss: 0.2377 - val_acc: 0.9170
Epoch 8/10 - 2216/2216 - 72s 32ms/step - loss: 0.2163 - acc: 0.9240 - val_loss: 0.2586 - val_acc: 0.8962
Epoch 9/10 - 2216/2216 - 68s 31ms/step - loss: 0.1992 - acc: 0.9301 - val_loss: 0.3247 - val_acc: 0.8782
Epoch 10/10 - 2216/2216 - 66s 30ms/step - loss: 0.1929 - acc: 0.9323 - val_loss: 0.2203 - val_acc: 0.9215

**Dataset 2:**
Train on 979 samples, validate on 244 samples
Epoch 1/10 - 979/979 - 30s 30ms/step - loss: 0.6753 - acc: 0.5822 - val_loss: 0.6565 - val_acc: 0.5902
Epoch 2/10 - 979/979 - 26s 27ms/step - loss: 0.6529 - acc: 0.6083 - val_loss: 0.6458 - val_acc: 0.6148
Epoch 3/10 - 979/979 - 27s 27ms/step - loss: 0.6397 - acc: 0.6236 - val_loss: 0.6509 - val_acc: 0.6414
Epoch 4/10 - 979/979 - 28s 29ms/step - loss: 0.6212 - acc: 0.6517 - val_loss: 0.6299 - val_acc: 0.6557
Epoch 5/10 - 979/979 - 25s 26ms/step - loss: 0.6036 - acc: 0.6844 - val_loss: 0.6260 - val_acc: 0.6639
Epoch 6/10 - 979/979 - 25s 26ms/step - loss: 0.5836 - acc: 0.6900 - val_loss: 0.6180 - val_acc: 0.6865
Epoch 7/10 - 979/979 - 25s 26ms/step - loss: 0.5789 - acc: 0.6685 - val_loss: 0.6406 - val_acc: 0.6537
Epoch 8/10 - 979/979 - 25s 26ms/step - loss: 0.5672 - acc: 0.7058 - val_loss: 0.6218 - val_acc: 0.6824
Epoch 9/10 - 979/979 - 26s 26ms/step - loss: 0.5305 - acc: 0.7523 - val_loss: 0.6387 - val_acc: 0.6537

Epoch 10/10 - 979/979 - 25s 26ms/step - loss: 0.5049 - acc: 0.7600 - val_loss: 0.7123 - val_acc: 0.6619

**Dataset 3:**
Train on 2594 samples, validate on 648 samples
Epoch 1/10 - 2594/2594 - 60s 23ms/step - loss: 0.5779 - acc: 0.6581 - val_loss: 0.5306 - val_acc: 0.6952
Epoch 2/10 - 2594/2594 - 57s 22ms/step - loss: 0.4566 - acc: 0.7614 - val_loss: 0.4639 - val_acc: 0.7793
Epoch 3/10 - 2594/2594 - 57s 22ms/step - loss: 0.4106 - acc: 0.8013 - val_loss: 0.4706 - val_acc: 0.7446
Epoch 4/10 - 2594/2594 - 59s 23ms/step - loss: 0.4057 - acc: 0.8076 - val_loss: 0.5082 - val_acc: 0.7153
Epoch 5/10 - 2594/2594 - 57s 22ms/step - loss: 0.3896 - acc: 0.8117 - val_loss: 0.6564 - val_acc: 0.7554
Epoch 6/10 - 2594/2594 - 58s 22ms/step - loss: 0.3924 - acc: 0.8138 - val_loss: 0.4359 - val_acc: 0.8002
Epoch 7/10 - 2594/2594 - 57s 22ms/step - loss: 0.3532 - acc: 0.8371 - val_loss: 0.4176 - val_acc: 0.8040
Epoch 8/10 - 2594/2594 - 56s 22ms/step - loss: 0.3344 - acc: 0.8479 - val_loss: 0.4726 - val_acc: 0.8063
Epoch 9/10 - 2594/2594 - 58s 22ms/step - loss: 0.3169 - acc: 0.8574 - val_loss: 0.4169 - val_acc: 0.7924
Epoch 10/10 - 2594/2594 - 57s 22ms/step - loss: 0.3077 - acc: 0.8568 - val_loss: 0.4107 - val_acc: 0.7955

*Double GRU*

**Dataset 1:**
Train on 2216 samples, validate on 554 samples
Epoch 1/10 - 2216/2216 - 148s 67ms/step - loss: 0.5566 - acc: 0.6983 - val_loss: 0.4373 - val_acc: 0.7816
Epoch 2/10 - 2216/2216 - 135s 61ms/step - loss: 0.4059 - acc: 0.8105 - val_loss: 0.3566 - val_acc: 0.8186
Epoch 3/10 - 2216/2216 - 139s 63ms/step - loss: 0.2607 - acc: 0.8879 - val_loss: 0.3159 - val_acc: 0.8899
Epoch 4/10 - 2216/2216 - 136s 62ms/step - loss: 0.3108 - acc: 0.8791 - val_loss: 0.2079 - val_acc: 0.9350
Epoch 5/10 - 2216/2216 - 90s 40ms/step - loss: 0.1621 - acc: 0.9429 - val_loss: 0.2016 - val_acc: 0.9206
Epoch 6/10 - 2216/2216 - 63s 28ms/step - loss: 0.1468 - acc: 0.9533 - val_loss: 0.1898 - val_acc: 0.9386
Epoch 7/10 - 2216/2216 - 93s 42ms/step - loss: 0.1122 - acc: 0.9605 - val_loss: 0.1792 - val_acc: 0.9350
Epoch 8/10 - 2216/2216 - 102s 46ms/step - loss: 0.1056 - acc: 0.9585 - val_loss: 0.2645 - val_acc: 0.9070
Epoch 9/10 - 2216/2216 - 99s 45ms/step - loss: 0.0946 - acc: 0.9625 - val_loss: 0.1451 - val_acc: 0.9495
Epoch 10/10 - 2216/2216 - 112s 50ms/step - loss: 0.0738 - acc: 0.9707 - val_loss: 0.1333 - val_acc: 0.9513

**Dataset 2:**

Train on 979 samples, validate on 244 samples
Epoch 1/10 - 979/979 - 28s 28ms/step - loss: 0.6741 - acc: 0.5695 - val_loss: 0.6400 - val_acc: 0.6168
Epoch 2/10 - 979/979 - 25s 25ms/step - loss: 0.6456 - acc: 0.6256 - val_loss: 0.6344 - val_acc: 0.6250
Epoch 3/10 - 979/979 - 25s 25ms/step - loss: 0.6281 - acc: 0.6404 - val_loss: 0.6610 - val_acc: 0.6270
Epoch 4/10 - 979/979 - 25s 25ms/step - loss: 0.6057 - acc: 0.6619 - val_loss: 0.6450 - val_acc: 0.6537
Epoch 5/10 - 979/979 - 25s 25ms/step - loss: 0.5716 - acc: 0.7028 - val_loss: 0.6592 - val_acc: 0.6537
Epoch 6/10 - 979/979 - 25s 25ms/step - loss: 0.5555 - acc: 0.7079 - val_loss: 0.6752 - val_acc: 0.6332
Epoch 7/10 - 979/979 - 25s 25ms/step - loss: 0.5042 - acc: 0.7625 - val_loss: 0.7287 - val_acc: 0.5840
Epoch 8/10 - 979/979 - 25s 25ms/step - loss: 0.4681 - acc: 0.7993 - val_loss: 0.8457 - val_acc: 0.5861
Epoch 9/10 - 979/979 - 25s 25ms/step - loss: 0.4193 - acc: 0.8136 - val_loss: 0.8739 - val_acc: 0.5410
Epoch 10/10 - 979/979 - 25s 25ms/step - loss: 0.3667 - acc: 0.8504 - val_loss: 0.9597 - val_acc: 0.6004

**Dataset 3:**
Train on 2594 samples, validate on 648 samples
Epoch 1/10 - 2594/2594 - 102s 39ms/step - loss: 0.5832 - acc: 0.6598 - val_loss: 0.4851 - val_acc: 0.7546
Epoch 2/10 - 2594/2594 - 98s 38ms/step - loss: 0.4705 - acc: 0.7537 - val_loss: 0.4287 - val_acc: 0.7878
Epoch 3/10 - 2594/2594 - 98s 38ms/step - loss: 0.4047 - acc: 0.7978 - val_loss: 0.3416 - val_acc: 0.8295
Epoch 4/10 - 2594/2594 - 97s 37ms/step - loss: 0.3899 - acc: 0.8184 - val_loss: 0.3606 - val_acc: 0.8210
Epoch 5/10 - 2594/2594 - 97s 37ms/step - loss: 0.3549 - acc: 0.8364 - val_loss: 0.3969 - val_acc: 0.7963
Epoch 6/10 - 2594/2594 - 97s 38ms/step - loss: 0.3502 - acc: 0.8410 - val_loss: 0.3771 - val_acc: 0.8133
Epoch 7/10 - 2594/2594 - 97s 38ms/step - loss: 0.3209 - acc: 0.8583 - val_loss: 0.3864 - val_acc: 0.8063
Epoch 8/10 - 2594/2594 - 98s 38ms/step - loss: 0.3126 - acc: 0.8614 - val_loss: 0.3515 - val_acc: 0.8372
Epoch 9/10 - 2594/2594 - 97s 38ms/step - loss: 0.2803 - acc: 0.8741 - val_loss: 0.3940 - val_acc: 0.8094
Epoch 10/10 - 2594/2594 - 97s 37ms/step - loss: 0.2573 - acc: 0.8851 - val_loss: 0.3913 - val_acc: 0.8218

*GRU + Dropout*

**Dataset 1:**
Train on 2216 samples, validate on 554 samples
Epoch 1/10 - 2216/2216 - 54s 24ms/step - loss: 0.5697 - acc: 0.6832 - val_loss: 0.4561 - val_acc: 0.7762

Epoch 2/10 - 2216/2216 - 53s 24ms/step - loss: 0.4120 - acc: 0.8089 - val_loss: 0.3012 - val_acc: 0.8899
Epoch 3/10 - 2216/2216 - 49s 22ms/step - loss: 0.2985 - acc: 0.8883 - val_loss: 0.4100 - val_acc: 0.8159
Epoch 4/10 - 2216/2216 - 49s 22ms/step - loss: 0.2612 - acc: 0.9066 - val_loss: 0.2736 - val_acc: 0.9034
Epoch 5/10 - 2216/2216 - 48s 22ms/step - loss: 0.2112 - acc: 0.9285 - val_loss: 0.3041 - val_acc: 0.9007
Epoch 6/10 - 2216/2216 - 51s 23ms/step - loss: 0.2609 - acc: 0.9075 - val_loss: 0.2511 - val_acc: 0.9025
Epoch 7/10 - 2216/2216 - 55s 25ms/step - loss: 0.2031 - acc: 0.9296 - val_loss: 0.2510 - val_acc: 0.9061
Epoch 8/10 - 2216/2216 - 48s 22ms/step - loss: 0.1821 - acc: 0.9341 - val_loss: 0.3400 - val_acc: 0.8556
Epoch 9/10 - 2216/2216 - 37s 17ms/step - loss: 0.2255 - acc: 0.9264 - val_loss: 0.2284 - val_acc: 0.9224
Epoch 10/10 - 2216/2216 - 62s 28ms/step - loss: 0.1606 - acc: 0.9488 - val_loss: 0.1929 - val_acc: 0.9305

**Dataset 2:**
Train on 979 samples, validate on 244 samples
Epoch 1/10 - 979/979 - 30s 31ms/step - loss: 0.6769 - acc: 0.5659 - val_loss: 0.6524 - val_acc: 0.6352
Epoch 2/10 - 979/979 - 26s 27ms/step - loss: 0.6484 - acc: 0.6078 - val_loss: 0.6566 - val_acc: 0.6311
Epoch 3/10 - 979/979 - 26s 27ms/step - loss: 0.6294 - acc: 0.6445 - val_loss: 0.6582 - val_acc: 0.6352
Epoch 4/10 - 979/979 - 26s 27ms/step - loss: 0.5966 - acc: 0.7012 - val_loss: 0.6522 - val_acc: 0.6537
Epoch 5/10 - 979/979 - 26s 26ms/step - loss: 0.5882 - acc: 0.6803 - val_loss: 0.6739 - val_acc: 0.6168
Epoch 6/10 - 979/979 - 26s 26ms/step - loss: 0.5949 - acc: 0.6731 - val_loss: 0.6306 - val_acc: 0.6660
Epoch 7/10 - 979/979 - 26s 27ms/step - loss: 0.5606 - acc: 0.7472 - val_loss: 0.6343 - val_acc: 0.6578
Epoch 8/10 - 979/979 - 26s 26ms/step - loss: 0.5494 - acc: 0.7482 - val_loss: 0.6467 - val_acc: 0.6824
Epoch 9/10 - 979/979 - 26s 26ms/step - loss: 0.5293 - acc: 0.7564 - val_loss: 0.6350 - val_acc: 0.6742
Epoch 10/10 - 979/979 - 26s 27ms/step - loss: 0.5161 - acc: 0.7559 - val_loss: 0.6395 - val_acc: 0.6844

**Dataset 3:**
Train on 2216 samples, validate on 554 samples
Epoch 1/10 - 2216/2216 - 54s 24ms/step - loss: 0.5697 - acc: 0.6832 - val_loss: 0.4561 - val_acc: 0.7762
Epoch 2/10 - 2216/2216 - 53s 24ms/step - loss: 0.4120 - acc: 0.8089 - val_loss: 0.3012 - val_acc: 0.8899
Epoch 3/10 - 2216/2216 - 49s 22ms/step - loss: 0.2985 - acc: 0.8883 - val_loss: 0.4100 - val_acc: 0.8159

Epoch 4/10 - 2216/2216 - 49s 22ms/step - loss: 0.2612 - acc: 0.9066 - val_loss: 0.2736 - val_acc: 0.9034
Epoch 5/10 - 2216/2216 - 48s 22ms/step - loss: 0.2112 - acc: 0.9285 - val_loss: 0.3041 - val_acc: 0.9007
Epoch 6/10 - 2216/2216 - 51s 23ms/step - loss: 0.2609 - acc: 0.9075 - val_loss: 0.2511 - val_acc: 0.9025
Epoch 7/10 - 2216/2216 - 55s 25ms/step - loss: 0.2031 - acc: 0.9296 - val_loss: 0.2510 - val_acc: 0.9061
Epoch 8/10 - 2216/2216 - 48s 22ms/step - loss: 0.1821 - acc: 0.9341 - val_loss: 0.3400 - val_acc: 0.8556
Epoch 9/10 - 2216/2216 - 37s 17ms/step - loss: 0.2255 - acc: 0.9264 - val_loss: 0.2284 - val_acc: 0.9224
Epoch 10/10 - 2216/2216 - 62s 28ms/step - loss: 0.1606 - acc: 0.9488 - val_loss: 0.1929 - val_acc: 0.9305

*LSTM Bidirectional*

**Dataset 1:**
Train on 2216 samples, validate on 554 samples
Epoch 1/10 - 2216/2216 - 48s 22ms/step - loss: 0.6151 - acc: 0.6264 - val_loss: 0.4690 - val_acc: 0.8060
Epoch 2/10 - 2216/2216 - 47s 21ms/step - loss: 0.3433 - acc: 0.8633 - val_loss: 0.2370 - val_acc: 0.9206
Epoch 3/10 - 2216/2216 - 47s 21ms/step - loss: 0.3179 - acc: 0.8766 - val_loss: 0.3443 - val_acc: 0.8529
Epoch 4/10 - 2216/2216 - 46s 21ms/step - loss: 0.2662 - acc: 0.8962 - val_loss: 0.3123 - val_acc: 0.8628
Epoch 5/10 - 2216/2216 - 47s 21ms/step - loss: 0.1761 - acc: 0.9413 - val_loss: 0.1956 - val_acc: 0.9332
Epoch 6/10 - 2216/2216 - 89s 40ms/step - loss: 0.3663 - acc: 0.8355 - val_loss: 0.3608 - val_acc: 0.8727
Epoch 7/10 - 2216/2216 - 116s 52ms/step - loss: 0.2910 - acc: 0.9084 - val_loss: 0.2417 - val_acc: 0.9269
Epoch 8/10 - 2216/2216 - 124s 56ms/step - loss: 0.2058 - acc: 0.9319 - val_loss: 0.3107 - val_acc: 0.8827
Epoch 9/10 - 2216/2216 - 106s 48ms/step - loss: 0.1946 - acc: 0.9380 - val_loss: 0.2584 - val_acc: 0.9070
Epoch 10/10 - 2216/2216 - 106s 48ms/step - loss: 0.1943 - acc: 0.9330 - val_loss: 0.2132 - val_acc: 0.9224

**Dataset 2:**
Train on 979 samples, validate on 244 samples
Epoch 1/10 - 979/979 - 23s 23ms/step - loss: 0.6731 - acc: 0.5689 - val_loss: 0.6678 - val_acc: 0.6025
Epoch 2/10 - 979/979 - 23s 23ms/step - loss: 0.6500 - acc: 0.5889 - val_loss: 0.6486 - val_acc: 0.5984
Epoch 3/10 - 979/979 - 22s 22ms/step - loss: 0.6311 - acc: 0.6287 - val_loss: 0.6357 - val_acc: 0.6578
Epoch 4/10 - 979/979 - 24s 24ms/step - loss: 0.6323 - acc: 0.6394 - val_loss: 0.6436 - val_acc: 0.6230
Epoch 5/10 - 979/979 - 24s 24ms/step - loss: 0.6158 - acc: 0.6680 - val_loss: 0.6399 - val_acc: 0.6721

Epoch 6/10 - 979/979 - 23s 23ms/step - loss: 0.6026 - acc: 0.6716 - val_loss: 0.6369 - val_acc: 0.6926
Epoch 7/10 - 979/979 - 20s 20ms/step - loss: 0.5711 - acc: 0.7155 - val_loss: 0.6101 - val_acc: 0.6598
Epoch 8/10 - 979/979 - 20s 20ms/step - loss: 0.5673 - acc: 0.7109 - val_loss: 0.6289 - val_acc: 0.6947
Epoch 9/10 - 979/979 - 20s 20ms/step - loss: 0.5748 - acc: 0.6966 - val_loss: 0.6299 - val_acc: 0.6803
Epoch 10/10 - 979/979 - 20s 20ms/step - loss: 0.5468 - acc: 0.7508 - val_loss: 0.6072 - val_acc: 0.7234

**Dataset 3:**
Train on 2594 samples, validate on 648 samples
Epoch 1/10 - 2594/2594 - 74s 29ms/step - loss: 0.6162 - acc: 0.6332 - val_loss: 0.4812 - val_acc: 0.7685
Epoch 2/10 - 2594/2594 - 69s 27ms/step - loss: 0.4705 - acc: 0.7741 - val_loss: 0.4253 - val_acc: 0.8156
Epoch 3/10 - 2594/2594 - 69s 27ms/step - loss: 0.5116 - acc: 0.7456 - val_loss: 0.4651 - val_acc: 0.7716
Epoch 4/10 - 2594/2594 - 70s 27ms/step - loss: 0.4536 - acc: 0.7724 - val_loss: 0.4034 - val_acc: 0.8194
Epoch 5/10 - 2594/2594 - 70s 27ms/step - loss: 0.4711 - acc: 0.7274 - val_loss: 0.4943 - val_acc: 0.7454
Epoch 6/10 - 2594/2594 - 70s 27ms/step - loss: 0.4882 - acc: 0.7440 - val_loss: 0.4616 - val_acc: 0.7639
Epoch 7/10 - 2594/2594 - 70s 27ms/step - loss: 0.4668 - acc: 0.7513 - val_loss: 0.4414 - val_acc: 0.7870
Epoch 8/10 - 2594/2594 - 70s 27ms/step - loss: 0.4202 - acc: 0.7901 - val_loss: 0.4122 - val_acc: 0.8225
Epoch 9/10 - 2594/2594 - 70s 27ms/step - loss: 0.4268 - acc: 0.7984 - val_loss: 0.4097 - val_acc: 0.8079
Epoch 10/10 - 2594/2594 - 70s 27ms/step - loss: 0.3903 - acc: 0.8146 - val_loss: 0.3964 - val_acc: 0.8171

*LSTM Bidirectional + Dropout*

**Dataset 1:**
Train on 2216 samples, validate on 554 samples
Epoch 1/10 - 2216/2216 - 79s 36ms/step - loss: 0.6361 - acc: 0.6094 - val_loss: 0.5212 - val_acc: 0.7924
Epoch 2/10 - 2216/2216 - 55s 25ms/step - loss: 0.4363 - acc: 0.7994 - val_loss: 0.3462 - val_acc: 0.8403
Epoch 3/10 - 2216/2216 - 55s 25ms/step - loss: 0.2954 - acc: 0.8858 - val_loss: 0.4498 - val_acc: 0.8014
Epoch 4/10 - 2216/2216 - 83s 37ms/step - loss: 0.2545 - acc: 0.9104 - val_loss: 0.2354 - val_acc: 0.9106
Epoch 5/10 - 2216/2216 - 90s 41ms/step - loss: 0.1808 - acc: 0.9355 - val_loss: 0.1621 - val_acc: 0.9422
Epoch 6/10 - 2216/2216 - 88s 40ms/step - loss: 0.1670 - acc: 0.9438 - val_loss: 0.2523 - val_acc: 0.9116
Epoch 7/10 - 2216/2216 - 89s 40ms/step - loss: 0.1621 - acc: 0.9454 - val_loss: 0.1655 - val_acc: 0.9449

Epoch 8/10 - 2216/2216 - 94s 42ms/step - loss: 0.2086 - acc: 0.9276 - val_loss: 0.1940 - val_acc: 0.9422
Epoch 9/10 - 2216/2216 - 92s 41ms/step - loss: 0.1855 - acc: 0.9404 - val_loss: 0.1577 - val_acc: 0.9495
Epoch 10/10 - 2216/2216 - 103s 47ms/step - loss: 0.1487 - acc: 0.9497 - val_loss: 0.1340 - val_acc: 0.9522

**Dataset 2:**
Train on 979 samples, validate on 244 samples
Epoch 1/10 - 979/979 - 48s 49ms/step - loss: 0.6706 - acc: 0.5832 - val_loss: 0.6766 - val_acc: 0.5779
Epoch 2/10 - 979/979 - 40s 41ms/step - loss: 0.6425 - acc: 0.6021 - val_loss: 0.6677 - val_acc: 0.5779
Epoch 3/10 - 979/979 - 40s 41ms/step - loss: 0.6941 - acc: 0.6384 - val_loss: 0.6910 - val_acc: 0.5779
Epoch 4/10 - 979/979 - 40s 41ms/step - loss: 0.6413 - acc: 0.6272 - val_loss: 0.6634 - val_acc: 0.6414
Epoch 5/10 - 979/979 - 38s 39ms/step - loss: 0.6222 - acc: 0.6762 - val_loss: 0.6999 - val_acc: 0.5779
Epoch 6/10 - 979/979 - 35s 36ms/step - loss: 0.6498 - acc: 0.6037 - val_loss: 0.6628 - val_acc: 0.6455
Epoch 7/10 - 979/979 - 35s 36ms/step - loss: 0.6059 - acc: 0.7048 - val_loss: 0.6471 - val_acc: 0.6660
Epoch 8/10 - 979/979 - 35s 36ms/step - loss: 0.5987 - acc: 0.6788 - val_loss: 0.7114 - val_acc: 0.5861
Epoch 9/10 - 979/979 - 35s 36ms/step - loss: 0.6124 - acc: 0.6624 - val_loss: 0.6651 - val_acc: 0.6025
Epoch 10/10 - 979/979 - 35s 36ms/step - loss: 0.5941 - acc: 0.7125 - val_loss: 0.6669 - val_acc: 0.6393

**Dataset 3:**
Train on 2594 samples, validate on 648 samples
Epoch 1/10 - 2594/2594 - 116s 45ms/step - loss: 0.6239 - acc: 0.6239 - val_loss: 0.5130 - val_acc: 0.7392
Epoch 2/10 - 2594/2594 - 105s 40ms/step - loss: 0.4985 - acc: 0.7373 - val_loss: 0.4759 - val_acc: 0.7546
Epoch 3/10 - 2594/2594 - 105s 40ms/step - loss: 0.4357 - acc: 0.7758 - val_loss: 0.4396 - val_acc: 0.7739
Epoch 4/10 - 2594/2594 - 105s 40ms/step - loss: 0.4306 - acc: 0.8005 - val_loss: 0.9335 - val_acc: 0.6289
Epoch 5/10 - 2594/2594 - 105s 40ms/step - loss: 0.5702 - acc: 0.6939 - val_loss: 0.5154 - val_acc: 0.7215
Epoch 6/10 - 2594/2594 - 108s 42ms/step - loss: 0.4834 - acc: 0.7513 - val_loss: 0.4871 - val_acc: 0.7446
Epoch 7/10 - 2594/2594 - 109s 42ms/step - loss: 0.4443 - acc: 0.7824 - val_loss: 0.5192 - val_acc: 0.7400
Epoch 8/10 - 2594/2594 - 110s 42ms/step - loss: 0.4394 - acc: 0.7839 - val_loss: 0.4273 - val_acc: 0.8009
Epoch 9/10 - 2594/2594 - 114s 44ms/step - loss: 0.3951 - acc: 0.8082 - val_loss: 0.4474 - val_acc: 0.7716

Epoch 10/10 - 2594/2594 - 109s 42ms/step - loss: 0.3902 - acc: 0.8150 - val_loss: 0.4276 - val_acc: 0.7971

## Appendix C Training Results for Additional Word Embedding Models

*Various Custom Word Embedding Sizes*

**Top 500 Words:**

Without stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4 - 58620/58620 - 102s 2ms/step - loss: 0.5257 - acc: 0.7625 - val_loss: 0.5080 - val_acc: 0.7743
Epoch 2/4 - 58620/58620 - 99s 2ms/step - loss: 0.5018 - acc: 0.7756 - val_loss: 0.5016 - val_acc: 0.7750
Epoch 3/4 - 58620/58620 - 99s 2ms/step - loss: 0.4947 - acc: 0.7791 - val_loss: 0.4970 - val_acc: 0.7788
Epoch 4/4 - 58620/58620 - 100s 2ms/step - loss: 0.4895 - acc: 0.7824 - val_loss: 0.4963 - val_acc: 0.7797
14655/14655 - 6s 391us/step
Test score: 0.4962786031985437
Test accuracy: 0.7797338792302428

With stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4
58620/58620 - 262s 4ms/step - loss: 0.5212 - acc: 0.7652 - val_loss: 0.5238 - val_acc: 0.7701
Epoch 2/4
58620/58620 - 320s 5ms/step - loss: 0.4965 - acc: 0.7793 - val_loss: 0.5164 - val_acc: 0.7694
Epoch 3/4
58620/58620 - 325s 6ms/step - loss: 0.4874 - acc: 0.7840 - val_loss: 0.5058 - val_acc: 0.7716
Epoch 4/4
58620/58620 - 323s 6ms/step - loss: 0.4812 - acc: 0.7878 - val_loss: 0.5041 - val_acc: 0.7730
14655/14655 - 20s 1ms/step
Test score: 0.5040943488819287
Test accuracy: 0.773046741742642

**Top 1000 Words:**

Without stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4
58620/58620 - 100s 2ms/step - loss: 0.5097 - acc: 0.7719 - val_loss: 0.4997 - val_acc: 0.7761
Epoch 2/4
58620/58620 - 130s 2ms/step - loss: 0.4810 - acc: 0.7870 - val_loss: 0.4957 - val_acc: 0.7800
Epoch 3/4

58620/58620 - 168s 3ms/step - loss: 0.4697 - acc: 0.7930 - val_loss: 0.4924 - val_acc: 0.7799
Epoch 4/4
58620/58620 - 169s 3ms/step - loss: 0.4593 - acc: 0.7994 - val_loss: 0.4909 - val_acc: 0.7795
14655/14655 - 10s 686us/step
Test score: 0.49087402938942337
Test accuracy: 0.7795291709476915

With stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4
58620/58620 - 273s 5ms/step - loss: 0.5108 - acc: 0.7725 - val_loss: 0.5003 - val_acc: 0.7836
Epoch 2/4
58620/58620 - 322s 5ms/step - loss: 0.4789 - acc: 0.7888 - val_loss: 0.4967 - val_acc: 0.7806
Epoch 3/4
58620/58620 - 326s 6ms/step - loss: 0.4685 - acc: 0.7952 - val_loss: 0.4836 - val_acc: 0.7861
Epoch 4/4
58620/58620 - 323s 6ms/step - loss: 0.4563 - acc: 0.8019 - val_loss: 0.4911 - val_acc: 0.7845
14655/14655 - 15s 1ms/step
Test score: 0.4911303585969072
Test accuracy: 0.7845104059559702

**Top 1500 Words:**

Without stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4
58620/58620 - 169s 3ms/step - loss: 0.5080 - acc: 0.7730 - val_loss: 0.4851 - val_acc: 0.7887
Epoch 2/4
58620/58620 - 170s 3ms/step - loss: 0.4719 - acc: 0.7941 - val_loss: 0.4792 - val_acc: 0.7885
Epoch 3/4
58620/58620 - 124s 2ms/step - loss: 0.4562 - acc: 0.8023 - val_loss: 0.4865 - val_acc: 0.7897
Epoch 4/4
58620/58620 - 88s 2ms/step - loss: 0.4410 - acc: 0.8097 - val_loss: 0.4833 - val_acc: 0.7898
14655/14655 - 5s 353us/step
Test score: 0.48325673144085746
Test accuracy: 0.7897645854819801

With stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4

58620/58620 - 715s 12ms/step - loss: 0.5094 - acc: 0.7715 - val_loss: 0.4851 - val_acc: 0.7852
Epoch 2/4
58620/58620 - 493s 8ms/step - loss: 0.4719 - acc: 0.7930 - val_loss: 0.4901 - val_acc: 0.7848
Epoch 3/4
58620/58620 - 323s 6ms/step - loss: 0.4571 - acc: 0.8008 - val_loss: 0.4731 - val_acc: 0.7907
Epoch 4/4
58620/58620 - 329s 6ms/step - loss: 0.4426 - acc: 0.8086 - val_loss: 0.4719 - val_acc: 0.7950
14655/14655 - 19s 1ms/step
Test score: 0.471925805368264
Test accuracy: 0.7950187648737728

**Top 2000 Words:**
Without stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4
58620/58620 - 384s 7ms/step - loss: 0.5056 - acc: 0.7731 - val_loss: 0.4870 - val_acc: 0.7929
Epoch 2/4
58620/58620 - 407s 7ms/step - loss: 0.4672 - acc: 0.7945 - val_loss: 0.4711 - val_acc: 0.7952
Epoch 3/4
58620/58620 - 540s 9ms/step - loss: 0.4476 - acc: 0.8049 - val_loss: 0.4783 - val_acc: 0.7933
Epoch 4/4
58620/58620 - 760s 13ms/step - loss: 0.4289 - acc: 0.8154 - val_loss: 0.4896 - val_acc: 0.7940
14655/14655 - 43s 3ms/step
Test score: 0.489629916559507
Test accuracy: 0.7939952234854188

With stopwords:
Epoch 1/4
58620/58620 - 734s 13ms/step - loss: 0.5044 - acc: 0.7749 - val_loss: 0.4934 - val_acc: 0.7829
Epoch 2/4
58620/58620 - 415s 7ms/step - loss: 0.4636 - acc: 0.7973 - val_loss: 0.4692 - val_acc: 0.7896
Epoch 3/4
58620/58620 - 320s 5ms/step - loss: 0.4467 - acc: 0.8072 - val_loss: 0.4788 - val_acc: 0.7851
Epoch 4/4
58620/58620 - 303s 5ms/step - loss: 0.4301 - acc: 0.8168 - val_loss: 0.4783 - val_acc: 0.7857
14655/14655 - 6s 393us/step
Test score: 0.47832598543053234
Test accuracy: 0.7856704196682647

**Top 2500 Words:**
Without stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4
58620/58620 - 404s 7ms/step - loss: 0.5043 - acc: 0.7749 - val_loss: 0.4969 - val_acc: 0.7853
Epoch 2/4
58620/58620 - 410s 7ms/step - loss: 0.4598 - acc: 0.7988 - val_loss: 0.4790 - val_acc: 0.7907
Epoch 3/4
58620/58620 - 600s 10ms/step - loss: 0.4390 - acc: 0.8110 - val_loss: 0.4814 - val_acc: 0.7876
Epoch 4/4
58620/58620 - 706s 12ms/step - loss: 0.4186 - acc: 0.8217 - val_loss: 0.4879 - val_acc: 0.7861
14655/14655 - 24s 2ms/step
Test score: 0.48794393906289096
Test accuracy: 0.7860798361845612

With stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4
58620/58620 - 315s 5ms/step - loss: 0.5036 - acc: 0.7758 - val_loss: 0.4918 - val_acc: 0.7801
Epoch 2/4
58620/58620 - 319s 5ms/step - loss: 0.4583 - acc: 0.8005 - val_loss: 0.4812 - val_acc: 0.7896
Epoch 3/4
58620/58620 - 325s 6ms/step - loss: 0.4385 - acc: 0.8110 - val_loss: 0.4822 - val_acc: 0.7842
Epoch 4/4
58620/58620 - 279s 5ms/step - loss: 0.4188 - acc: 0.8209 - val_loss: 0.4918 - val_acc: 0.7868
14655/14655 - 10s 679us/step
Test score: 0.491755419932954
Test accuracy: 0.7868304332463418

**Top 3000 Words:**
Without stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4
58620/58620 - 390s 7ms/step - loss: 0.4988 - acc: 0.7776 - val_loss: 0.4894 - val_acc: 0.7885
Epoch 2/4
58620/58620 - 408s 7ms/step - loss: 0.4527 - acc: 0.8038 - val_loss: 0.4802 - val_acc: 0.7889
Epoch 3/4
58620/58620 - 551s 9ms/step - loss: 0.4301 - acc: 0.8155 - val_loss: 0.4850 - val_acc: 0.7857
Epoch 4/4

58620/58620 - 730s 12ms/step - loss: 0.4055 - acc: 0.8293 - val_loss: 0.5110 - val_acc: 0.7872
14655/14655 - 43s 3ms/step
Test score: 0.5110026398664121
Test accuracy: 0.7872398498887212

With stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4
58620/58620 - 714s 12ms/step - loss: 0.5022 - acc: 0.7759 - val_loss: 0.4832 - val_acc: 0.7831
Epoch 2/4
58620/58620 - 492s 8ms/step - loss: 0.4560 - acc: 0.8024 - val_loss: 0.4859 - val_acc: 0.7834
Epoch 3/4
58620/58620 - 323s 6ms/step - loss: 0.4310 - acc: 0.8155 - val_loss: 0.4752 - val_acc: 0.7871
Epoch 4/4
58620/58620 - 327s 6ms/step - loss: 0.4078 - acc: 0.8287 - val_loss: 0.4976 - val_acc: 0.7816
14655/14655 - 20s 1ms/step
Test score: 0.4976094910590895
Test accuracy: 0.7815762538464148

**Top 3500 Words:**
Without stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4
58620/58620 - 402s 7ms/step - loss: 0.4974 - acc: 0.7814 - val_loss: 0.4843 - val_acc: 0.7919
Epoch 2/4
58620/58620 - 408s 7ms/step - loss: 0.4473 - acc: 0.8068 - val_loss: 0.4711 - val_acc: 0.7896
Epoch 3/4
58620/58620 - 583s 10ms/step - loss: 0.4200 - acc: 0.8216 - val_loss: 0.4885 - val_acc: 0.7905
Epoch 4/4
58620/58620 - 717s 12ms/step - loss: 0.3921 - acc: 0.8351 - val_loss: 0.5249 - val_acc: 0.7875
14655/14655 - 30s 2ms/step
Test score: 0.524882550104939
Test accuracy: 0.7875127942803695

With stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4
58620/58620 - 323s 6ms/step - loss: 0.4993 - acc: 0.7782 - val_loss: 0.4894 - val_acc: 0.7891
Epoch 2/4
58620/58620 - 321s 5ms/step - loss: 0.4510 - acc: 0.8052 - val_loss: 0.4798 - val_acc: 0.7932

Epoch 3/4
58620/58620 - 327s 6ms/step - loss: 0.4262 - acc: 0.8179 - val_loss: 0.4790 - val_acc: 0.7907
Epoch 4/4
58620/58620 - 248s 4ms/step - loss: 0.4023 - acc: 0.8306 - val_loss: 0.4906 - val_acc: 0.7913
14655/14655 - 5s 355us/step
Test score: 0.4906210924061196
Test accuracy: 0.791334015645496

**Top 4000 Words:**
Without stopwords:
Train on 58620 samples, validate on 14655 samples
Epoch 1/4
58620/58620 - 395s 7ms/step - loss: 0.4980 - acc: 0.7784 - val_loss: 0.4751 - val_acc: 0.7944
Epoch 2/4
58620/58620 - 408s 7ms/step - loss: 0.4451 - acc: 0.8085 - val_loss: 0.4731 - val_acc: 0.7963
Epoch 3/4
58620/58620 - 570s 10ms/step - loss: 0.4159 - acc: 0.8227 - val_loss: 0.4828 - val_acc: 0.7925
Epoch 4/4
58620/58620 - 726s 12ms/step - loss: 0.3878 - acc: 0.8365 - val_loss: 0.5269 - val_acc: 0.7905
14655/14655 - 34s 2ms/step
Test score: 0.5269294022456324
Test accuracy: 0.7905151824868202

With stopwords:
Epoch 1/4
58620/58620 - 718s 12ms/step - loss: 0.4981 - acc: 0.7790 - val_loss: 0.4679 - val_acc: 0.7963
Epoch 2/4
58620/58620 - 489s 8ms/step - loss: 0.4482 - acc: 0.8054 - val_loss: 0.4771 - val_acc: 0.7944
Epoch 3/4
58620/58620 - 321s 5ms/step - loss: 0.4206 - acc: 0.8205 - val_loss: 0.4705 - val_acc: 0.7998
Epoch 4/4
58620/58620 - 327s 6ms/step - loss: 0.3956 - acc: 0.8334 - val_loss: 0.4814 - val_acc: 0.7962
14655/14655 - 20s 1ms/step
Test score: 0.4814114273895952
Test accuracy: 0.7961787785209922

*Deeper RNN models*

**GRU Custom Word Embedding 1. Training**
Use tf.cast instead.
Train on 58620 samples, validate on 14655 samples

Epoch 1/4 - 58620/58620 - 956s 16ms/step - loss: 0.5195 - acc: 0.7674 - val_loss: 0.4987 - val_acc: 0.7745
Epoch 2/4 - 58620/58620 - 449s 8ms/step - loss: 0.4708 - acc: 0.7942 - val_loss: 0.4907 - val_acc: 0.7803
Epoch 3/4 - 58620/58620 - 449s 8ms/step - loss: 0.4636 - acc: 0.7979 - val_loss: 0.4979 - val_acc: 0.7752
Epoch 4/4 - 58620/58620 - 474s 8ms/step - loss: 0.4517 - acc: 0.8037 - val_loss: 0.4909 - val_acc: 0.7788

**GRU Custom Word Embedding 2. Training**
Train on 58620 samples, validate on 14655 samples
Epoch 1/4 - 58620/58620 - 972s 17ms/step - loss: 0.5221 - acc: 0.7653 - val_loss: 0.4965 - val_acc: 0.7821
Epoch 2/4 - 58620/58620 - 451s 8ms/step - loss: 0.4755 - acc: 0.7919 - val_loss: 0.4848 - val_acc: 0.7870
Epoch 3/4 - 58620/58620 - 450s 8ms/step - loss: 0.4614 - acc: 0.7997 - val_loss: 0.4822 - val_acc: 0.7881
Epoch 4/4 - 58620/58620 - 474s 8ms/step - loss: 0.4598 - acc: 0.7991 - val_loss: 0.4878 - val_acc: 0.7835

**GRU Pre-Trained Word Embedding 1. Training**
Train on 2594 samples, validate on 648 samples
Epoch 1/10 - 2594/2594 - 99s 38ms/step - loss: 0.5894 - acc: 0.6370 - val_loss: 0.5056 - val_acc: 0.7253
Epoch 2/10 - 2594/2594 - 97s 37ms/step - loss: 0.4889 - acc: 0.7355 - val_loss: 0.4542 - val_acc: 0.7577
Epoch 3/10 - 2594/2594 - 97s 37ms/step - loss: 0.4233 - acc: 0.7911 - val_loss: 0.4509 - val_acc: 0.7708
Epoch 4/10 - 2594/2594 - 96s 37ms/step - loss: 0.3936 - acc: 0.8094 - val_loss: 0.4190 - val_acc: 0.8017
Epoch 5/10 - 2594/2594 - 96s 37ms/step - loss: 0.3666 - acc: 0.8259 - val_loss: 0.3816 - val_acc: 0.8349
Epoch 6/10 - 2594/2594 - 97s 37ms/step - loss: 0.3642 - acc: 0.8283 - val_loss: 0.4349 - val_acc: 0.7909
Epoch 7/10 - 2594/2594 - 96s 37ms/step - loss: 0.3819 - acc: 0.8051 - val_loss: 0.3901 - val_acc: 0.8148
Epoch 8/10 - 2594/2594 - 96s 37ms/step - loss: 0.3420 - acc: 0.8342 - val_loss: 0.3805 - val_acc: 0.8187
Epoch 9/10 - 2594/2594 - 96s 37ms/step - loss: 0.3281 - acc: 0.8466 - val_loss: 0.3871 - val_acc: 0.8187
Epoch 10/10 - 2594/2594 - 96s 37ms/step - loss: 0.3002 - acc: 0.8601 - val_loss: 0.3801 - val_acc: 0.8372

**GRU Pre-Trained Word Embedding 2. Training**
Train on 2594 samples, validate on 648 samples
Epoch 1/10 - 2594/2594 - 100s 39ms/step - loss: 0.5730 - acc: 0.6615 - val_loss: 0.5270 - val_acc: 0.7176
Epoch 2/10 - 2594/2594 - 98s 38ms/step - loss: 0.4445 - acc: 0.7783 - val_loss: 0.4893 - val_acc: 0.7323
Epoch 3/10 - 2594/2594 - 97s 37ms/step - loss: 0.4065 - acc: 0.7988 - val_loss: 0.4315 - val_acc: 0.7909

Epoch 4/10 - 2594/2594 - 98s 38ms/step - loss: 0.3706 - acc: 0.8242 - val_loss: 0.4288 - val_acc: 0.7971
Epoch 5/10 - 2594/2594 - 98s 38ms/step - loss: 0.3831 - acc: 0.8202 - val_loss: 0.4294 - val_acc: 0.7948
Epoch 6/10 - 2594/2594 - 97s 38ms/step - loss: 0.3499 - acc: 0.8400 - val_loss: 0.4380 - val_acc: 0.7731
Epoch 7/10 - 2594/2594 - 96s 37ms/step - loss: 0.3387 - acc: 0.8404 - val_loss: 0.4346 - val_acc: 0.8094
Epoch 8/10 - 2594/2594 - 97s 37ms/step - loss: 0.4202 - acc: 0.7808 - val_loss: 0.5180 - val_acc: 0.7153
Epoch 9/10 - 2594/2594 - 97s 38ms/step - loss: 0.4340 - acc: 0.7847 - val_loss: 0.4996 - val_acc: 0.7361
Epoch 10/10 - 2594/2594 - 94s 36ms/step - loss: 0.4003 - acc: 0.8063 - val_loss: 0.4986 - val_acc: 0.7438

**Double GRU Custom Word Embedding 1. Training**
Train on 58620 samples, validate on 14655 samples
Epoch 1/4 - 58620/58620 - 1460s 25ms/step - loss: 0.5235 - acc: 0.7645 - val_loss: 0.4985 - val_acc: 0.7795
Epoch 2/4 - 58620/58620 - 1732s 30ms/step - loss: 0.4921 - acc: 0.7781 - val_loss: 0.4806 - val_acc: 0.7873
Epoch 3/4 - 58620/58620 - 1667s 28ms/step - loss: 0.4644 - acc: 0.7959 - val_loss: 0.4769 - val_acc: 0.7898
Epoch 4/4 - 58620/58620 - 1669s 28ms/step - loss: 0.4559 - acc: 0.8025 - val_loss: 0.4754 - val_acc: 0.7899

**Double GRU Custom Word Embedding 2. Training**
Train on 58620 samples, validate on 14655 samples
Epoch 1/4 - 58620/58620 - 1462s 25ms/step - loss: 0.5270 - acc: 0.7602 - val_loss: 0.5163 - val_acc: 0.7627
Epoch 2/4 - 58620/58620 - 1737s 30ms/step - loss: 0.4838 - acc: 0.7843 - val_loss: 0.4854 - val_acc: 0.7835
Epoch 3/4 - 58620/58620 - 1675s 29ms/step - loss: 0.4618 - acc: 0.7976 - val_loss: 0.4798 - val_acc: 0.7881
Epoch 4/4 - 58620/58620 - 1671s 29ms/step - loss: 0.4498 - acc: 0.8056 - val_loss: 0.4785 - val_acc: 0.7889

**Double GRU Pre-Trained Word Embedding 1. Training**
Train on 2594 samples, validate on 648 samples
Epoch 1/4 - 2594/2594 - 78s 30ms/step - loss: 0.5776 - acc: 0.6727 - val_loss: 0.4415 - val_acc: 0.7886
Epoch 2/4 - 2594/2594 - 146s 56ms/step - loss: 0.4607 - acc: 0.7660 - val_loss: 0.4608 - val_acc: 0.7577
Epoch 3/4 - 2594/2594 - 159s 61ms/step - loss: 0.4145 - acc: 0.8009 - val_loss: 0.4013 - val_acc: 0.7924
Epoch 4/4 - 2594/2594 - 152s 58ms/step - loss: 0.3773 - acc: 0.8217 - val_loss: 0.3959 - val_acc: 0.7894

**Double GRU Pre-Trained Word Embedding 2. Training**
Train on 2594 samples, validate on 648 samples

Epoch 1/4 - 2594/2594 - 155s 60ms/step - loss: 0.5624 - acc: 0.7093 - val_loss: 0.5097 - val_acc: 0.7215
Epoch 2/4 - 2594/2594 - 143s 55ms/step - loss: 0.4514 - acc: 0.7681 - val_loss: 0.4919 - val_acc: 0.7238
Epoch 3/4 - 2594/2594 - 146s 56ms/step - loss: 0.3862 - acc: 0.8086 - val_loss: 0.4610 - val_acc: 0.7577
Epoch 4/4 - 2594/2594 - 151s 58ms/step - loss: 0.3716 - acc: 0.8155 - val_loss: 0.4148 - val_acc: 0.7840

**LSTM Custom Word Embedding 1. Training**
Train on 58620 samples, validate on 14655 samples
Epoch 1/4 - 58620/58620 - 1128s 19ms/step - loss: 0.5211 - acc: 0.7660 - val_loss: 0.5048 - val_acc: 0.7796
Epoch 2/4 - 58620/58620 - ETA: 0s - loss: 0.4845 - acc: 0.783 - 585s 10ms/step - loss: 0.4845 - acc: 0.7834 - val_loss: 0.5227 - val_acc: 0.7589
Epoch 3/4 - 58620/58620 - 841s 14ms/step - loss: 0.4839 - acc: 0.7848 - val_loss: 0.4924 - val_acc: 0.7807
Epoch 4/4 - 58620/58620 - 834s 14ms/step - loss: 0.4666 - acc: 0.7969 - val_loss: 0.4899 - val_acc: 0.7819

**LSTM Custom Word Embedding 2. Training**
Train on 58620 samples, validate on 14655 samples
Epoch 1/4 - 58620/58620 - 1149s 20ms/step - loss: 0.5252 - acc: 0.7627 - val_loss: 0.4896 - val_acc: 0.7847
Epoch 2/4 - 58620/58620 - ETA: 0s - loss: 0.4894 - acc: 0.781 - 587s 10ms/step - loss: 0.4894 - acc: 0.7817 - val_loss: 0.4832 - val_acc: 0.7876
Epoch 3/4 - 58620/58620 - 847s 14ms/step - loss: 0.4753 - acc: 0.7900 - val_loss: 0.4815 - val_acc: 0.7879
Epoch 4/4 - 58620/58620 - 842s 14ms/step - loss: 0.4708 - acc: 0.7917 - val_loss: 0.4842 - val_acc: 0.7908

**LSTM Pre-Trained Word Embedding 1. Training**
Train on 2594 samples, validate on 648 samples
Epoch 1/4 - 2594/2594 - 127s 49ms/step - loss: 0.3446 - acc: 0.8387 - val_loss: 0.4305 - val_acc: 0.7840
Epoch 2/4 - 2594/2594 - 120s 46ms/step - loss: 0.3925 - acc: 0.7859 - val_loss: 0.4876 - val_acc: 0.7415
Epoch 3/4 - 2594/2594 - 119s 46ms/step - loss: 0.4082 - acc: 0.7936 - val_loss: 0.4406 - val_acc: 0.7708
Epoch 4/4 - 2594/2594 - 124s 48ms/step - loss: 0.3680 - acc: 0.8267 - val_loss: 0.4674 - val_acc: 0.7662

**LSTM Pre-Trained Word Embedding 2. Training**

Train on 2594 samples, validate on 648 samples
Epoch 1/4 - 2594/2594 - 127s 49ms/step - loss: 0.3242 - acc: 0.8529 - val_loss: 0.4200 - val_acc: 0.8025
Epoch 2/4 - 2594/2594 - 119s 46ms/step - loss: 0.3009 - acc: 0.8604 - val_loss: 0.4506 - val_acc: 0.7793
Epoch 3/4 - 2594/2594 - 119s 46ms/step - loss: 0.2837 - acc: 0.8741 - val_loss: 0.4175 - val_acc: 0.8040

Epoch 4/4 - 2594/2594 - 124s 48ms/step - loss: 0.2627 - acc: 0.8843 - val_loss: 0.4693 - val_acc: 0.7986

**Double LSTM Custom Word Embedding 1. Training**
Train on 58620 samples, validate on 14655 samples
Epoch 1/4 - 58620/58620 - 2153s 37ms/step - loss: 0.5186 - acc: 0.7671 - val_loss: 0.4936 - val_acc: 0.7782
Epoch 2/4 - 58620/58620 - 2072s 35ms/step - loss: 0.4815 - acc: 0.7879 - val_loss: 0.4965 - val_acc: 0.7790
Epoch 3/4 - 58620/58620 - 1633s 28ms/step - loss: 0.4697 - acc: 0.7939 - val_loss: 0.4896 - val_acc: 0.7810
Epoch 4/4 - 58620/58620 - 1320s 23ms/step - loss: 0.4587 - acc: 0.7996 - val_loss: 0.4839 - val_acc: 0.7883

**Double LSTM Custom Word Embedding 2. Training**
Train on 58620 samples, validate on 14655 samples
Epoch 1/4 - 58620/58620 - 2189s 37ms/step - loss: 0.5328 - acc: 0.7554 - val_loss: 0.4933 - val_acc: 0.7793
Epoch 2/4 - 58620/58620 - 2109s 36ms/step - loss: 0.4855 - acc: 0.7832 - val_loss: 0.4821 - val_acc: 0.7915
Epoch 3/4 - 58620/58620 - 1626s 28ms/step - loss: 0.4697 - acc: 0.7931 - val_loss: 0.4823 - val_acc: 0.7881
Epoch 4/4 - 58620/58620 - 1309s 22ms/step - loss: 0.4631 - acc: 0.7979 - val_loss: 0.4799 - val_acc: 0.7913

**Double LSTM Pre-Trained Word Embedding 1. Training**
Train on 2594 samples, validate on 648 samples
Epoch 1/4 - 2594/2594 - 244s 94ms/step - loss: 0.5824 - acc: 0.6887 - val_loss: 0.5354 - val_acc: 0.7361
Epoch 2/4 - 2594/2594 - 162s 62ms/step - loss: 0.4500 - acc: 0.7939 - val_loss: 0.4010 - val_acc: 0.8040
Epoch 3/4 - 2594/2594 - 160s 62ms/step - loss: 0.3918 - acc: 0.8198 - val_loss: 0.3708 - val_acc: 0.8187
Epoch 4/4 - 2594/2594 - 159s 61ms/step - loss: 0.3972 - acc: 0.8042 - val_loss: 0.4470 - val_acc: 0.7623

**Double LSTM Pre-Trained Word Embedding 2. Training**
Train on 2594 samples, validate on 648 samples
Epoch 1/4 - 2594/2594 - 166s 64ms/step - loss: 0.6029 - acc: 0.6582 - val_loss: 0.4774 - val_acc: 0.7731
Epoch 2/4 - 2594/2594 - 183s 70ms/step - loss: 0.4556 - acc: 0.7708 - val_loss: 0.3950 - val_acc: 0.8110
Epoch 3/4 - 2594/2594 - 177s 68ms/step - loss: 0.4200 - acc: 0.7955 - val_loss: 0.3603 - val_acc: 0.8403
Epoch 4/4 - 2594/2594 - 162s 62ms/step - loss: 0.3842 - acc: 0.8190 - val_loss: 0.3579 - val_acc: 0.8380

*Final Trainings*

**CNN**

1. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/2
2594/2594 - 34s 13ms/step - loss: 0.5817 - acc: 0.6874 - val_loss: 0.3896 - val_acc: 0.8187
Epoch 2/2
2594/2594 - 33s 13ms/step - loss: 0.3589 - acc: 0.8311 - val_loss: 0.3051 - val_acc: 0.8588

2. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/2
2594/2594 - 31s 12ms/step - loss: 0.5995 - acc: 0.6843 - val_loss: 0.4206 - val_acc: 0.8148
Epoch 2/2
2594/2594 - 30s 12ms/step - loss: 0.3807 - acc: 0.8250 - val_loss: 0.3220 - val_acc: 0.8457

3. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/2
2594/2594 - 57s 22ms/step - loss: 0.3557 - acc: 0.8308 - val_loss: 0.3083 - val_acc: 0.8542
Epoch 2/2
2594/2594 - 57s 22ms/step - loss: 0.2916 - acc: 0.8770 - val_loss: 0.3039 - val_acc: 0.8688

4. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/2
2594/2594 - 38s 14ms/step - loss: 0.5744 - acc: 0.6987 - val_loss: 0.3911 - val_acc: 0.8140
Epoch 2/2
2594/2594 - 37s 14ms/step - loss: 0.3503 - acc: 0.8338 - val_loss: 0.3325 - val_acc: 0.8380

**GRU + Dropout**

1. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/6
2594/2594 - 93s 36ms/step - loss: 0.5794 - acc: 0.6663 - val_loss: 0.5211 - val_acc: 0.7207
Epoch 2/6
2594/2594 - 37s 14ms/step - loss: 0.4618 - acc: 0.7645 - val_loss: 0.4597 - val_acc: 0.7693
Epoch 3/6
2594/2594 - 37s 14ms/step - loss: 0.4403 - acc: 0.7841 - val_loss: 0.4715 - val_acc: 0.7639
Epoch 4/6

2594/2594 - 38s 15ms/step - loss: 0.4096 - acc: 0.8067 - val_loss: 0.4516 - val_acc: 0.7631
Epoch 5/6
2594/2594 - 41s 16ms/step - loss: 0.3955 - acc: 0.8057 - val_loss: 0.4586 - val_acc: 0.7531
Epoch 6/6
2594/2594 - 41s 16ms/step - loss: 0.3723 - acc: 0.8217 - val_loss: 0.4454 - val_acc: 0.7878

2. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/6
2594/2594 - 42s 16ms/step - loss: 0.5965 - acc: 0.6372 - val_loss: 0.5304 - val_acc: 0.7114
Epoch 2/6
2594/2594 - 45s 17ms/step - loss: 0.4850 - acc: 0.7452 - val_loss: 0.5080 - val_acc: 0.7083
Epoch 3/6
2594/2594 - 41s 16ms/step - loss: 0.4555 - acc: 0.7675 - val_loss: 0.4674 - val_acc: 0.7623
Epoch 4/6
2594/2594 - 63s 24ms/step - loss: 0.4257 - acc: 0.7843 - val_loss: 0.4535 - val_acc: 0.7639
Epoch 5/6
2594/2594 - 65s 25ms/step - loss: 0.3935 - acc: 0.8113 - val_loss: 0.4334 - val_acc: 0.7793
Epoch 6/6
2594/2594 - 65s 25ms/step - loss: 0.3679 - acc: 0.8261 - val_loss: 0.4217 - val_acc: 0.7971

3. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/6
2594/2594 - 68s 26ms/step - loss: 0.5861 - acc: 0.6486 - val_loss: 0.5218 - val_acc: 0.7191
Epoch 2/6
2594/2594 - 62s 24ms/step - loss: 0.4794 - acc: 0.7496 - val_loss: 0.4705 - val_acc: 0.7508
Epoch 3/6
2594/2594 - 62s 24ms/step - loss: 0.4181 - acc: 0.7936 - val_loss: 0.4195 - val_acc: 0.8040
Epoch 4/6
2594/2594 - 50s 19ms/step - loss: 0.3929 - acc: 0.8096 - val_loss: 0.4551 - val_acc: 0.7639
Epoch 5/6
2594/2594 - 36s 14ms/step - loss: 0.4071 - acc: 0.7976 - val_loss: 0.4669 - val_acc: 0.7577
Epoch 6/6
2594/2594 - 36s 14ms/step - loss: 0.3840 - acc: 0.8152 - val_loss: 0.4292 - val_acc: 0.8009

4. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/6
2594/2594 - 44s 17ms/step - loss: 0.5897 - acc: 0.6505 - val_loss: 0.5211 - val_acc: 0.7215
Epoch 2/6
2594/2594 - 47s 18ms/step - loss: 0.4784 - acc: 0.7537 - val_loss: 0.4866 - val_acc: 0.7284
Epoch 3/6
2594/2594 - 72s 28ms/step - loss: 0.4595 - acc: 0.7554 - val_loss: 0.4373 - val_acc: 0.7693
Epoch 4/6
2594/2594 - 71s 27ms/step - loss: 0.4178 - acc: 0.7922 - val_loss: 0.4382 - val_acc: 0.7901
Epoch 5/6
2594/2594 - 72s 28ms/step - loss: 0.3894 - acc: 0.8167 - val_loss: 0.3934 - val_acc: 0.8140
Epoch 6/6
2594/2594 - 69s 27ms/step - loss: 0.3955 - acc: 0.8157 - val_loss: 0.3992 - val_acc: 0.8002

**LSTM + Dropout**

1. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/6
2594/2594 - 89s 34ms/step - loss: 0.5877 - acc: 0.6586 - val_loss: 0.5043 - val_acc: 0.7438
Epoch 2/6
2594/2594 - 86s 33ms/step - loss: 0.4705 - acc: 0.7639 - val_loss: 0.4621 - val_acc: 0.7616
Epoch 3/6
2594/2594 - 87s 34ms/step - loss: 0.4279 - acc: 0.7818 - val_loss: 0.4817 - val_acc: 0.7461
Epoch 4/6
2594/2594 - 63s 24ms/step - loss: 0.3908 - acc: 0.8121 - val_loss: 0.4427 - val_acc: 0.7785
Epoch 5/6
2594/2594 - 48s 18ms/step - loss: 0.3955 - acc: 0.8036 - val_loss: 0.4648 - val_acc: 0.7284
Epoch 6/6
2594/2594 - 50s 19ms/step - loss: 0.4058 - acc: 0.7862 - val_loss: 0.4031 - val_acc: 0.7847

2. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/6
2594/2594 - 82s 31ms/step - loss: 0.5898 - acc: 0.6473 - val_loss: 0.4750 - val_acc: 0.7670
Epoch 2/6

2594/2594 - 78s 30ms/step - loss: 0.4698 - acc: 0.7658 - val_loss: 0.4607 - val_acc: 0.7685
Epoch 3/6
2594/2594 - 64s 25ms/step - loss: 0.4556 - acc: 0.7643 - val_loss: 0.4440 - val_acc: 0.7809
Epoch 4/6
2594/2594 - 49s 19ms/step - loss: 0.4125 - acc: 0.7959 - val_loss: 0.4058 - val_acc: 0.7971
Epoch 5/6
2594/2594 - 54s 21ms/step - loss: 0.4173 - acc: 0.8067 - val_loss: 0.3935 - val_acc: 0.8272
Epoch 6/6
2594/2594 - 76s 29ms/step - loss: 0.4441 - acc: 0.7866 - val_loss: 0.4195 - val_acc: 0.8241

3. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/6
2594/2594 - 79s 31ms/step - loss: 0.4204 - acc: 0.7905 - val_loss: 0.3883 - val_acc: 0.8125
Epoch 2/6
2594/2594 - 78s 30ms/step - loss: 0.3840 - acc: 0.8192 - val_loss: 0.3872 - val_acc: 0.8133
Epoch 3/6
2594/2594 - 50s 19ms/step - loss: 0.3735 - acc: 0.8346 - val_loss: 0.3891 - val_acc: 0.8110
Epoch 4/6
2594/2594 - 42s 16ms/step - loss: 0.3644 - acc: 0.8337 - val_loss: 0.3690 - val_acc: 0.8302
Epoch 5/6
2594/2594 - 42s 16ms/step - loss: 0.3395 - acc: 0.8408 - val_loss: 0.4090 - val_acc: 0.7940
Epoch 6/6
2594/2594 - 43s 16ms/step - loss: 0.3424 - acc: 0.8398 - val_loss: 0.3667 - val_acc: 0.8256

4. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/6
2594/2594 - 81s 31ms/step - loss: 0.5868 - acc: 0.6559 - val_loss: 0.5129 - val_acc: 0.7137
Epoch 2/6
2594/2594 - 78s 30ms/step - loss: 0.4794 - acc: 0.7581 - val_loss: 0.4390 - val_acc: 0.7755
Epoch 3/6
2594/2594 - 77s 30ms/step - loss: 0.4713 - acc: 0.7483 - val_loss: 0.4662 - val_acc: 0.7685
Epoch 4/6
2594/2594 - 57s 22ms/step - loss: 0.4337 - acc: 0.7805 - val_loss: 0.3937 - val_acc: 0.8071
Epoch 5/6

2594/2594 - 78s 30ms/step - loss: 0.4199 - acc: 0.7995 - val_loss: 0.4025 - val_acc: 0.8102
Epoch 6/6
2594/2594 - 78s 30ms/step - loss: 0.3971 - acc: 0.8074 - val_loss: 0.3836 - val_acc: 0.8272

**LSTM Bidirectional**

1. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/6 - 2594/2594 - 51s 20ms/step - loss: 0.6219 - acc: 0.6523 - val_loss: 0.5210 - val_acc: 0.7423
Epoch 2/6 - 2594/2594 - 48s 19ms/step - loss: 0.4674 - acc: 0.7697 - val_loss: 0.4334 - val_acc: 0.7894
Epoch 3/6 - 2594/2594 - 48s 19ms/step - loss: 0.4025 - acc: 0.8098 - val_loss: 0.4000 - val_acc: 0.8133
Epoch 4/6 - 2594/2594 - 48s 19ms/step - loss: 0.4110 - acc: 0.8001 - val_loss: 0.3917 - val_acc: 0.8117
Epoch 5/6 - 2594/2594 - 48s 19ms/step - loss: 0.3930 - acc: 0.8153 - val_loss: 0.4010 - val_acc: 0.8156
Epoch 6/6 - 2594/2594 - 48s 19ms/step - loss: 0.3797 - acc: 0.8252 - val_loss: 0.3750 - val_acc: 0.8117

2. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/6 - 2594/2594- 98s 38ms/step - loss: 0.6177 - acc: 0.6357 - val_loss: 0.5281 - val_acc: 0.6944
Epoch 2/6 - 2594/2594- 99s 38ms/step - loss: 0.4573 - acc: 0.7901 - val_loss: 0.4350 - val_acc: 0.7917
Epoch 3/6 - 2594/2594- 99s 38ms/step - loss: 0.4189 - acc: 0.8009 - val_loss: 0.4354 - val_acc: 0.8009
Epoch 4/6 - 2594/2594- 95s 36ms/step - loss: 0.4316 - acc: 0.7899 - val_loss: 0.4395 - val_acc: 0.7917
Epoch 5/6 - 2594/2594- 94s 36ms/step - loss: 0.4864 - acc: 0.7286 - val_loss: 0.5286 - val_acc: 0.7315
Epoch 6/6 - 2594/2594- 94s 36ms/step - loss: 0.4921 - acc: 0.7463 - val_loss: 0.4867 - val_acc: 0.7569

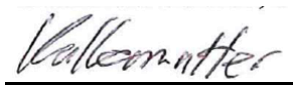
3. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/6 - 2594/2594- 105s 40ms/step - loss: 0.6200 - acc: 0.6342 - val_loss: 0.5056 - val_acc: 0.7554
Epoch 2/6 - 2594/2594- 99s 38ms/step - loss: 0.5154 - acc: 0.7114 - val_loss: 0.5134 - val_acc: 0.7423
Epoch 3/6 - 2594/2594- 99s 38ms/step - loss: 0.5084 - acc: 0.7388 - val_loss: 0.4871 - val_acc: 0.7508
Epoch 4/6 - 2594/2594- 95s 37ms/step - loss: 0.4889 - acc: 0.7438 - val_loss: 0.4789 - val_acc: 0.7554
Epoch 5/6 - 2594/2594- 95s 37ms/step - loss: 0.4703 - acc: 0.7540 - val_loss: 0.4675 - val_acc: 0.7569

Epoch 6/6 - 2594/2594- 95s 36ms/step - loss: 0.4416 - acc: 0.7795 - val_loss: 0.4394 - val_acc: 0.7878

4. Training
Train on 2594 samples, validate on 648 samples
Epoch 1/6 - 2594/2594- 54s 21ms/step - loss: 0.6145 - acc: 0.6345 - val_loss: 0.5224 - val_acc: 0.7585
Epoch 2/6 - 2594/2594- 51s 20ms/step - loss: 0.4677 - acc: 0.7704 - val_loss: 0.4252 - val_acc: 0.8117
Epoch 3/6 - 2594/2594- 51s 20ms/step - loss: 0.4306 - acc: 0.8001 - val_loss: 0.4413 - val_acc: 0.7878
Epoch 4/6 - 2594/2594- 51s 19ms/step - loss: 0.4102 - acc: 0.8082 - val_loss: 0.4131 - val_acc: 0.8264
Epoch 5/6 - 2594/2594- 50s 19ms/step - loss: 0.3961 - acc: 0.8211 - val_loss: 0.4231 - val_acc: 0.7870
Epoch 6/6 - 2594/2594- 50s 19ms/step - loss: 0.3877 - acc: 0.8225 - val_loss: 0.3836 - val_acc: 0.8272

## Declaration of Autorship

I herewith declare that this is my independent work written by me and using only admissible aides and no other sources than those given. I have marked as such, all passages, which have been taken literally or analogously from another source. I am aware that if this is not the case, the executive board of the university of applied sciences is entitled to rescind any qualifications awarded or any title bestowed based on this work.

Chur, 09.08.2019 

Place, Date Urban Kalbermatter

**Bisher erschienene Schriften**

Ergebnisse von Forschungsprojekten erscheinen jeweils in Form von Arbeitsberichten in Reihen. Sonstige Publikationen erscheinen in Form von alleinstehenden Schriften.

Derzeit gibt es in den Churer Schriften zur Informationswissenschaft folgende Reihen:
Reihe Berufsmarktforschung

Weitere Publikationen finden Sie unter folgendem Link:

https://www.fhgr.ch/fhgr/angewandte-zukunftstechnologien/schweizerisches-institut-fuer-informationswissenschaft-sii/publikationen/churer-schriften/

Churer Schriften zur Informationswissenschaft – Schrift 93
Herausgegeben von Wolfgang Semar
Silvana Rüfli
Die Usability von E-Book-Angeboten wissenschaftlicher Bibliotheken Eine Untersuchung am Beispiel der Universitätsbibliotheken
St. Gallen, Bern und Zürich
Chur, 2018
ISSN 1660-945X

Churer Schriften zur Informationswissenschaft – Schrift 94
Herausgegeben von Wolfgang Semar
Vera Knoll
Leichte Sprache in amtlichen Publikationen und Webseiten
Wie ernst nehmen Verwaltungen die Leichte Sprache in der deutschsprachigen Schweiz?
Chur, 2018
ISSN 1660-945X

Churer Schriften zur Informationswissenschaft – Schrift 95
Herausgegeben von Wolfgang Semar
Andrea Traber
Wie lernen studentische Bibliotheks-Nutzende und was macht für sie den optimalen Arbeitsplatz aus?
Eine Studie der Lernlandschaft der Universitätsbibliothek St. Gallen
Chur, 2018
ISSN 1660-945X

Churer Schriften zur Informationswissenschaft – Schrift 96
Herausgegeben von Wolfgang Semar
Irina Morell
„Für das Volk und durch das Volk?“
Bibliotheken als Gegenstand von Volksabstimmungen und Petitionen
Chur, 2018
ISSN 1660-945X

Churer Schriften zur Informationswissenschaft – Schrift 97
Herausgegeben von Wolfgang Semar
Monika Rohner
Betrachtung der Data Visualization Literacy in der angestrebten Schweizer Informationsgesellschaft
Chur, 2018
ISSN 1660-945X

Churer Schriften zur Informationswissenschaft – Schrift 98
Herausgegeben von Wolfgang Semar
Kirsten Scherer Auberson
Counteracting Concept Drift in Natural Language Classifiers: Proposal for an Automated Method
Chur, 2018
ISSN 1660-945X

Churer Schriften zur Informationswissenschaft – Schrift 99
Herausgegeben von Wolfgang Semar
Hanna Kummel
Enhancing Collaboration in Collaborative Problem-Solving with Conversational Agents
Chur, 2019
ISSN 1660-945X

Churer Schriften zur Informationswissenschaft – Schrift 100
Herausgegeben von Wolfgang Semar
Carina Burch
Community – eine Untersuchung was es im Kontext von allgemein-öffentlichen Bibliotheken bedeutet
Chur, 2019
ISSN 1660-945X

Churer Schriften zur Informationswissenschaft – Schrift 101
Herausgegeben von Wolfgang Semar
Reihe Berufsmarktforschung – Arbeitsbericht 8 Sharon Alt, Bernard Bekavac, Urs Dahinden Absolventenstudie 2017
Bachelorstudiengang Information Science, MAS Information Science, Masterstudienrichtung Information and Data Management
Chur, 2019
ISSN 1660-945X

Churer Schriften zur Informationswissenschaft – Schrift 102
Herausgegeben von Wolfgang Semar
Debora Greter
Wissensmanagement in der Lebensmittelindustrie
Konzept zur Integration von Wissensmanagement in bestehende Qualitäts- und Lebensmittelsicherheits-Managementsysteme
Chur, 2019
ISSN 1660-945X

Churer Schriften zur Informationswissenschaft – Schrift 103
Herausgegeben von Wolfgang Semar
Urban Kalbermatter
Deep learning for detecting integrity risks in text documents
Chur, 2019
ISSN 1660-945X

# Über die Informationswissenschaft der Fachhochschule Graubünden

Die Informationswissenschaft ist in der Schweiz noch ein relativ junger Lehr- und Forschungs- bereich. International weist diese Disziplin aber vor allem im anglo-amerikanischen Bereich eine jahrzehntelange Tradition auf. Die klassischen Bezeichnungen dort sind Information Science, Library Science oder Information Studies. Die Grundfragestellung der Informationswissenschaft liegt in der Betrachtung der Rolle und des Umgangs mit Information in allen ihren Ausprägungen und Medien sowohl in Wirtschaft und Gesellschaft. Die Informationswissenschaft wird in Chur integriert betrachtet.

Diese Sicht umfasst nicht nur die Teildisziplinen Bibliothekswissenschaft, Archivwissenschaft und Dokumentationswissenschaft. Auch neue Entwicklungen im Bereich Medienwirtschaft, Informa- tions- und Wissensmanagement und Big Data werden gezielt aufgegriffen und im Lehr- und Forschungsprogramm berücksichtigt.

Der Studiengang Informationswissenschaft wird seit 1998 als Vollzeitstudiengang in Chur ange- boten und seit 2002 als Teilzeit-Studiengang in Zürich. Seit 2010 rundet der Master of Science in Business Administration das Lehrangebot ab.

Der Arbeitsbereich Informationswissenschaft vereinigt Cluster von Forschungs-, Entwicklungs- und Dienstleistungspotenzialen in unterschiedlichen Kompetenzzentren:

- Information Management & Competitive Intelligence
- Collaborative Knowledge Management
- Information and Data Management
- Records Management
- Library Consulting
- Information Laboratory

Diese Kompetenzzentren werden im **Swiss Institute for Information Research** zusammen- gefasst.

# IMPRESSUM

Verlag & Anschrift

**Arbeitsbereich Informationswissenschaft**

FHGR – Fachhochschule Graubünden
University of Applied Sciences
Pulvermühlestrasse 57
CH – 7000 Chur

www.blog.fhgr.ch/dis

www.fhgr.ch

**ISSN 1660-945X**

Institutsleitung

Prof. Dr. Ingo Barkow

Telefon: +41 81 286 24 61

Email: ingo.barkow@fhgr.ch

Sekretariat

Telefon: +41 81 286 24 24

Fax: +41 81 286 24 00

Email: clarita.decurtins@fhgr.ch