Contents

Chapter 1

Introduction

Getting computers to understand human language is a complex task. :)
Introduction - not started
Prelims - in progress
Task definitions - needs section on POS tagging
Datset - SemEval - needs formatting and re-check
Machine Learning part - needs semi-supervised and self-training parts
validation - make sure we generalize well, not started yet
Neural Networks
need graph and soma reference
Optimization - not started
need reference to maximum loglikelihood
regularization - not started
Conv need structuring and formatting
Related Works - in progress
CNNs - need a small paragraph for each network since 2015
need semi-supervised section
Discussion - not started Conclusion - not started

1.1 Problem definition

1.2 Research Questions

I have formulated the following tentative research questions:

- What are the current state-of-the-art systems for relation classification with neural networks?

- Is it possible to simplify structures in state-of-the-art systems while maintaining (or increasing) the F1 metric?

Chapter 2

Preliminaries

This chapter provides the background information necessary understand the relation classification task and how the task can be solved with machine learning.

2.1 Task Definitions

Natural Language Processing (NLP) is a research area which combines linguistics and computer science with the goal of understanding human - or natural - language in various forms such as written text or audio recordings. We might separate the field further in smaller challenges like Natural Language Understanding (NLU) or Natural Language Generation (NLG), but for the rest of this report I will present all related tasks under the definition NLP.

Understanding human language in a computer usually involves building a structured model over the input language which the computer can understand and interpret. These structures can model low-level syntactic information about the structure of the language. An example of such as model is a *Part-Of-Speech (POS) tagger*, which labels each word token with a distinct category related to the definition of the word and its context. The output of a POS-tagger can be used as input in another task. An example of such a task is the *semantic classification task*, which captures higher-level semantic information. I will cover both definitions.

2.1.1 POS-tagging

TBD Given a sequence of written text as input we can label each word

" She soon had a stable of her own rescued hounds "

2.1.2 Semantic Relation Classification

With the output of the task defined in 2.1.1 we can now look for pairs of certain entities in sentences and extract their relation. Specifically, we can look for pairs of common nominals and identify a potential semantic relation between them. Common nominals are word(s) which act as a noun in a sentence. Identifying and categorizing these relations is called *semantic relation classification*. The difference between general relation classification and semantic relation classification is subtle and not well defined. The word 'semantic' is used because of the possible classes the relation classification task can output(15)(21). For the rest of the thesis, I will denote semantic relation classification as simply relation classification (RC).
The goal of the task is to output an ordered tuple which conveys meaning about the relationship between the marked nominals. Consider the example used above where the common nouns have been marked:

" She soon had a stable$_{e1}$of her own rescued hounds$_{e2}$"

A correct output in this example is the tuple Member-Collection(e2,e1). The label is intended to capture subtle semantic information about the nominals. A "stable" can mean a physical building which houses animals, but it can also mean a set of animals which are grouped together by human means. The Member-Collection tuple defines the word "stable" to have a specific meaning due to the relation to the other nominal.

The task can be more formally defined: Given an input sentence S and entities W_{e1} and W_{e2} which are taken from S, we define an input space $\mathcal{X}$ which is the set of all triples (S, W_{e1}, W_{e2}). We also define a set of labels $\mathcal{Y}$. These labels are probably different from dataset from dataset, but they define the space of the relations we are looking for. The relation classification task is to find a mapping $h : \mathcal{X} \mapsto \mathcal{Y}$. The correct mapping h^* is not well-defined. Generally the requirement for h is that some human evaluator must agree with the semantic relation which is given by h. When solving the task for a specific dataset, a test set can be provided with a given (sentence,label) set. The goal of RC is then to output the same labels as the human annotators.

A closely related task to RC is *relation extraction* (RE) . The actual difference between the two tasks are ill-defined and often confused because 1) they are both classification problems and 2) they both have the same inputs. For the sake of clarity in this thesis, I will define the RE task to include first the binary classification of whether an actual relation exists between inputs. This means that the RE tasks usually use unbalanced datasets, where the majority of the samples have no relation such as the ACE 2005 dataset(39).

Finally, RC assumes that the common nouns have always been correctly identified. This assumption, of course, does not hold if POS-tagging is done by a statistical algorithm. Incorrect labels in the output might lead to error propagation. A way to deal with this assumption is to include the marking of the common nominals in the RC task. This approach is called *end-to-end relation classification*, or *relation detection*. This approach lies beyond the scope of this thesis

2.2 Datasets

One major problem with the RC task is the lack of quality datasets that can be used in academia to measure the performance and robustness of freshly developed classifiers. Relations are both hard to define and classify, and the diversity of language usually requires that a large amount of samples are labeled before the dataset is useful. The types of relations which the model should classify also varies greatly. Relations are not easy to define generically and are sometimes even defined only for a specific research paper(10)(20). Finally, to use the dataset on the relation classification task, the dataset should be

- *Open*: The dataset should be publicly available so the research community can use it without too many restrictions, and claimed results on the dataset can be validated
- *Well-defined*: The format of the dataset should be well-defined to avoid large amounts of preprocessing which can have significant effect on the subsequent classification. It can also be useful if some official metric is defined for measuring the performance on the dataset. While the goal of a classification model may vary on the dataset, having an official tool helps the research community compare the results of their models.
- *Clean*: This requirement is specific to the relation classification task. The relation extraction task should cover gathering the input for the classification. As such, noisy and missing input misses the goal of the RC task.

Table 2.2
Label distribution for SemEval 2010 task 8

Relation	Distribution
Cause-Effect	12.4%
Component-Whole	11.7%
Entity-Destination	10.6%
Entity-Origin	9.1%
Product-Producer	8.8%
Member-Collection	8.6%
Message-Topic	8.4%
Content-Container	6.8%
Instrument-Agency	6.2%
Other	17.4%

Table 2.1
Train-test distribution for SemEval 2010 task 8

Samples	
Train	8000
Test	2717
Total	10717

- *Used*: Finally, the dataset is most useful if it has some traction in the research community, which creates a well-defined benchmark for the task.

2.2.1 SemEval 2010 Task 8

One dataset which shows the characteristics listed above is the dataset created for the 2010 SemEval task 10(21). This dataset defines a sentence classification task with relation labels which are semantically exclusive. The task is canonical for relation classification as the distribution of labels are fairly equal, which means that the entity detection step is assumed. Otherwise, there would a significant amount of samples in the dataset with no relation. Notice that the "Other" relation is distinct from a no-relation class. The Other class consists of samples where the type of the relation could not be determined though they still represent some relation.

In summary, solving the RC task requires us to find a general function which can label sentences by only knowing the entities and the sentence itself. To find this function we turn to a branch of computer science called *machine learning*.

2.3 Machine Learning

Modern solutions to the RC problem is almost always based on machine learning techniques. These techniques allow computers to solve problems by learning rules and patterns from data. An alternative to machine learning is hand-written rule-based systems - but these are hard to engineer and error-prone since language varies greatly and words have different meaning based on their context. In this section I will describe different learning problems and relate it to the RC problem.

2.3.1 Supervised and unsupervised learning

A common task in machine learning is a generalized version of the RC task: we want to learn a mapping $h : \mathcal{X} \mapsto \mathcal{Y}$ from a set of datapoints D_{train}. The nature of D_{train} and the range of $\mathcal{Y}$ defines the associated machine learning task(2). D_{train} can be points (x_i, y_i) where y_i is the given label for x_i, usually annotated by a human, or it can be unlabeled points (x_i) where no label is given. An algorithm that uses labels (x_i, y_i) to learn h places itself in the class of *supervised learning algorithms*, while algorithms that only use unlabeled data (x_i) are called *unsupervised learning algorithms*.

2.3.2 Learning problems

We can define a set of four problems by combining the input D_{train} with the range of $\mathcal{Y}$. If $\mathcal{Y}$ is discrete-valued and the input uses labeled data, we call h a *classification function*.

If $\mathcal{Y}$ is continuous, the task is called *regression*. If we have no labels from the input but Y is still discrete, we call the problem *clustering* as the algorithm usually must make its discrete values. And finally, if no labeled data is available and $\mathcal{Y}$ is continuous, h can be called a *density estimation function*, since we can interpret $\mathcal{Y}$ as being proportional to a probability distribution over $\mathcal{X}$.

2.3.3 Semi-supervised classification

We can look at the amount of data in D_{train} which have labels to define a broader category of algorithms that contains both supervised and unsupervised algorithms. With this perspective, supervised learning are algorithms that assume all data is labeled, while unsupervised assumes none. If we have *some* data which is labeled,

2.3.4 Self-Training

Parameters and Techniques

2.3.5 Metrics

To evaluate the performance of a RC function h, we can apply the function on a set of data $D_{test} = [(S_1, W_{1,e1}, W_{2,e2} \dots (S_n, W_{n,e1}, W_{n,e2}]$ in which the labels $h^*(D_{test}) = [y_1 \dots y_n]$ are known. Since the labels of the relations are discrete, a sample triple $(S_i, W_{i,e1}, W_{i,e2})$ is correct if $h(S_i, W_{i,e1}, W_{i,e2}) = y_i$. It is also useful to measure what label is predicted instead of the correct label when the classifier is wrong. By observing the prediction of each label we can construct a *confusion matrix* which shows predictions on one axis and the actual labels on the other. Below is shown an example of a confusion matrix for a 10-way classification problem:

Fig.???

From the positions in the confusion matrix predictions for each class Y_i are measured and put into four categories:

- A *true positive* (tp) are a sample $D_i \in D_{test}$ where $h^*(D_i) = Y_i$ and $h(D_i) = Y_i$.
- If $h^*(D_i) \neq Y_i$ and $h(D_i) = Y_i$, D_i is a *false positive* (fp).
- Conversely, if $h^*(D_i) \neq Y_i$ and $h(D_i) \neq Y_i$, D_i is a *true negative* (tn).
- Finally, if $h^*(D_i) = Y_i$ and $h(D_i) \neq Y_i$ D_i is a *false negative* (fn).

F1 Measure

The accuracy for a multi-class problem is defined with these terms over the number of classes m as $\frac{\sum_i^m tp(Y_i)}{n}$, but it is not very useful since it does not take the distribution of classes into account. In the relation extraction task, for example, the number of samples which have no relation will greatly outweigh the relevant samples. A classifier may be encouraged to simply label all samples as having no relation, which will yield a high accuracy. Instead, two measures which shows performance for each can be used. *Precision* (p) is defined as $\frac{tp}{tp+fp}$ and indicates how certain a classifier is that predictions for a class actually belong to that class. *Recall* (r) is defined as $\frac{tp}{tp+fn}$ and indicates how sensitive a classifier is to samples belonging to a certain class. Precision and recall are ends of a spectrum - a classifier can achieve maximum precision for a class by never predicting that class, but at the cost of recall. Likewise recall can be trivially obtained by cutting precision. To output a single number for a class which balances these two, the harmonic mean of precision and recall is defined as the *F1 score* $\frac{2*p*r}{p+r}$. An example of the measures are shown below, drawn from the confusion matrix:

Fig.???

And finally an averaging strategy for the individual F1 scores must be chosen to output a single score. By averaging each final f1 score we obtain the *macro* F1:

$$F1_{macro} \frac{\sum_i^m \frac{2*p_{Y_i}*r_{Y_i}}{p_{Y_i}+r_{Y_i}}}{m}.$$

Alternatively we can sum individual precision and recall values which will value larger classes higher and obtain the micro:

$$p_{micro} = \frac{\sum_i^m tp_{Y_i}}{\sum_i^m tp_{Y_i} + \sum_i^m fp_{Y_i}} r_{micro} = \frac{\sum_i^m tp_{Y_i}}{\sum_i^m tp_{Y_i} + \sum_i^m fn_{Y_i}} F1_{micro} = \frac{2 * p_{micro} * r_{micro}}{p_{micro} + r_{micro}}$$

For NLP tasks, the macro is often chosen as classes with low frequency usually are important. For example, if the RC task is extended to also include the detection of relations in natural text, the relations occur with low frequency and are important to classify correctly.

2.3.6 Validation

In the sections above I briefly described that the F1 is applied to a dataset D_{test}. Choosing D_{test} correctly is important for getting a good estimate of how well h will perform on unseen data. This is because measuring how well h is doing on D_{train} does not guarantee that it will do well on unseen data. Learned models are subject to *overfitting*.

Overfitting

Overfitting occurs when the chosen h has a high accuracy on the data on which it was trained, but low accuracy on unseen data. Why does overfitting occur? The main reasons for overfitting is:

- *Noisy data*: Real data has noise which can either be *deterministic* or *stochastic*. Deterministic noise is part of the data which cannot be modeled by h. When deterministic noise happens in the training data, h might see this as a pattern on which it can generalize when it is fact part of a different function than h. Stochastic noise happens because the process that generates the data might be error-prone. Relations which are classified by a human annotator might be wrong, or encoding errors might obscure some text from the system. (abu)
- *Target complexity*: The function h is learned only from the data. Due to noise, however, the training data can mislead the model which find h to think a more complex function is needed that better fits the data. When *capacity* of a model that finds h is high, the risk of overfitting increases because the model have more ways of choosing incorrectly. (**?** , p. 107). For neural networks detailed in section 2.4, the capacity is usually huge because the network have many layers and parameters.

Validation Set

Because of overfitting, it is necessary to measure h on a dataset which is not used to find h. First we create a test set D_test which must not be used *at all* to select h or the space from which h can be chosen. For the SemEval data, the test set is already designated. Next, to measure the effectiveness of h during the learning process, we split the remaining data D into D_{train} and D_{val}. D_{val} should be a size which makes it a reliable estimate of how h will perform on D_{test}. A common size is 10% of D.

We can now use our metrics on D_{val} to estimate how well h will perform on D_{test} without actually touching D_{test}. Another way to prevent overfitting is to reduce the capacity of the model used to find h. This technique is called *regularization*. Regularization for neural networks is detailed in subsection 2.4.3

2.4 Deep Feedforward Neural Networks

There are many different algorithms to choose from when trying to learn a relation classifier. A widely used datastructure is the *neural network*, which have been used extensively in recent years with significant results in several research areas including NLP. The recent work on neural networks in NLP and specifically RC will be covered in **??**. This section covers the definition of a deep feedforward neural network and how it is trained to approximate h through derivatives. Choosing the correct structure of the network that approximates h is a problem that often leads to the concept of *overfitting* (18, sec. 5.2). The final part of this section covers a group of *regularization* techniques that constrain the model to prevent this from happening.

2.4.1 Definition

A deep feedforward neural network is a datastructure which can be used to learn and represent a function $h : X \mapsto Y$ from a set of training data D_{train}. Usually, the term neural network also encompasses the algorithms which are used to build the datastructure and approximate the function h. A common way to describe neural networks is to describe each word:

The word 'network' is used because the datastructure represents a series of functions which are chained together to form a network. This layering of functions forms a directed graph $f(0) \mapsto f(1) \dots f(d-1) \mapsto f(d)$ which begins with the input space $\mathcal{X}$ and ends with the output $\mathcal{Y}$. The first layer $f(0)$ is called the *input layer*, while the final layer $f(d)$ is called the *output layer*. Layers in between are called *hidden layers*. The layers are called hidden because they are not defined by the training data and must be learned by the individual network(**?** , chapter 6). The number of hidden layers d is called the *depth* of the network.

The word 'feedforward' is used to describe that the network has no connections that leads back to previous layers. The information flows in one direction through the network when it is being activated. A network that has information from the output flowing back into the network is called a *recurrent neural net*. Recurrent nets and their usage in NLP is detailed in section 3.1.3

The word 'deep' is used when the number of hidden layers are more than zero. A network with no hidden layers - so that the input layer is directly connected to the output layer - is commonly called a *perceptron*. While it is true that a network with a single hidden layer can be used to approximate any function, it has been shown that increasing the depth greatly reduces the number of neurons needed in the network compared to a single layer network.

Consider the l'th layer in a network. A common notation for the output of a layer $f(l)$ in a neural network is to write it as the input for the next layer x_l The inputs for layer l is a vector output of the previous layer function denoted $x^l = f(l-1)$. We can apply an entire vector-to-vector function on the previous input:

$$x^l = f(x^l - 1)$$

However, we can also deconstruct x^l to its individual components:

$$x^l = [f_0^l; f_1^l \dots; f_n^l]^T$$

Each component f_i^l is a vector-to-scalar function that can be applied in parallel on the input vector. Each component is called a *neuron* or a *unit*. The length $n_l = |f(l)|$ is called the *width* and can also be seen as the number of neurons in the layer. For clarity we denote the width of the previous layer as $m = n_{l-1} = |x|$ The word 'neural' is used in neural networks because the way these components are constructed take inspiration from models of biological brains. Each component consists of a vector, which individual values determines how much information flows from one neuron f_i^{l-1} in the previous layer to a neuron in the next. The strength of these connections are called *weights*. All weights for a particular neuron are summed to form the *activation* of the neuron. The activation for a neuron f_i^l is then a linear combination of the inputs determined by the weights $w_{i0}^l, w_{i1}^l \ldots w_{in_{l-1}}^l$:

$$f_i^l = \sum_{j=0}^{|f(l-1)|} f_j^{l-1} * w_{ij}^l$$

Given the above definitions and the fact that $f(l)$ have n activations with m weights, we can write all the weights for a layer as a matrix $W^l \in \mathbb{R}^n \times m$ and the total activation of $f(l)$ as a matrix-vector product:

$$x^l = W^l x^l$$

Activation and Bias

A neuron in the brain contains an *axon* which is a transmitter that measures a potential in the *soma* which is modeled with the summations described above. When the potential in the soma is large enough, the axon delivers a signal to neurons it is connected to. We can model this thresholding by introducing a non-linear *activation function*, usually denoted σ, which "fires" when the input is large enough to pass a threshold. The functions usually squeeze the input into a certain interval which is analogous to neurons "firing" in the brain when they have received enough signal to pass a threshold. In order for the threshold to be specific to each neuron a *bias parameter* b is added to the transformation. We can now write the full output of a layer $f(l)$ once the activation and the bias have been applied:

$$f(l) = \sigma(W^l x^l) + b$$

Note that σ and b is applied element-wise to each neuron in l.

As I will show in subsection 2.4.2 the derivative of the activation function is an important factor in choosing which activation function to use.

The traditional activation function is the *logistic sigmoid* function, which is a smooth "S"-shaped function which centers around zero. The derivative is simple and easy to compute. However, the sigmoid suffers not being centered around zero, which is a problem for gradient-based optimization.(18, p. 66). To compensate for this problem, the *hyperbolic tangent* function is used, which is centered around zero. Both functions suffer from *saturation* which is also a problem for optimization (16). A third option is then the *rectified linear unit* (ReLU), which does not saturate. All three and their derivatives are shown below:

Fig.???

From the above components it is possible to define any function h by a network defined by its depth, width, activation functions and its weights. The depth, width and activation function are chosen by the designer of the network while the specific weights are learned by the optimization algorithm. We can parameterize h to include the weights $w = [w^i \ldots w^d]$ by writing $h(x, w)$, where x is the input to the first layer in the network.

2.4.2 Optimization

The question remains: how do we use the network $h(x, w)$ to approximate $h : \mathcal{X} \mapsto Y$? To do this, we need to construct a way of measuring of how well $h(x, w)$ predicts the samples in D_{train}. The measures shown in section 2.3.5 cannot reveal any information about *why* the chosen setting of w predicts incorrectly. Instead we need a more general function which directly depends on w which, when minimized, indirectly maximizes the F1 score. This measure is usually called an *objective function* or *loss function*

Objective function

Consider a supervised learning problem where h maps input x to a single discrete output label $y_i \in \mathcal{Y}$. We can construct the output layer $y(x)$ of the network $h(x,w)$ such that each label in Y has a corresponding output neuron. To find out which y_i we should choose from the network predictions we inspect the activations of the neurons. One way of interpreting the predictions is as a conditional probability distribution over the classes in $\mathcal{Y}$ given x: $P_{pred}(Y = y|X = x)$. To make sure that the output of the neurons correspond to a probability distribution over Y, we can use a different activation function which normalizes the output. A common example of such a function is the *softmax*. Given the output layer described above the soft-max is defined as

$$\sigma(y_i) = \frac{e^{y_i}}{\sum_{j=0}^{K} e^{y_j}}$$

where K is the total number of neurons in the output layer.

Since we know the label for x in advance because x is in D_{train}, we can also construct the probability distribution P_{true} over Y by setting the probability for the label to 1 and all other probabilities to 0. Such a distribution is also called a *one-hot encoding*. We can use how well the predicted distribution aligns with the true distribution of D_train with a concept borrowed from information theory called the *cross-entropy loss*. The loss is most useful when it is averaged over a number of samples N:

$$Ł(P_{pred}, P_{true}) = -\frac{1}{N}\sum_{i}^{N}\sum_{j}^{K} P^{i}_{pred}(y_j) * log(P^{i}_{true}(y_j))$$

In other words, the cross-entropy loss measures a distance between the predicted and true probability of a sample i belonging to a class y_j. This cross-entropy is equivalent to the (average negative log-likelihood), which provides additional theoretical grounding for using the loss as a measure of how well $h(x,w)$ predicts D_{train}. It is important to notice that this loss function is defined only over the training data and therefore is not an exact measure of how well h actually performs on new data. TODO - REFERENCE shows that it is possible to relate the loss to the generalization error theoretically under the assumption that D_{train} is drawn from the same distribution as the new data. In practice, we can measure how well h performs empirically by using the validation techniques shown in subsection 2.3.6

Backpropagation

Since P_{pred} is subject to the chosen h in the neural network and P_{true} is given from D_{train}, we can write INSERT EQUALITY REF as a function of these: $Ł(h, D_{train})$. We now have a function subject to minimization with differential calculus. Since h is defined by its weights w (when the structure and the activation function of the network is fixed), we can apply an algorithm known as *backpropagation* to calculate how each weight in w contributes to Ł.

2.4.3 Regularization

Early Stopping

A non-intrusive way of limiting overfitting on the training set is to continually measure the performance of the classification function on a set of validation data for which we can measure the loss without updating the network. When the validation loss stops decreasing, we record the number of training iterations run on the network. This method is called *early stopping*. The validation data must be independent from the training data and the test data which is used to estimate the generalization error. Early stopping can be applied to any neural network. A drawback of the method is that the validation data is not used for training. A way to deal with this problem is to use the validation data to find the number of training iterations, and then re-train the network with all the available data. The full pseudocode for the algorithm is detailed in (**?** , p. 242).

Dropout

Another popular technique for regularization is *dropout* (36). A simple but powerful idea, dropout constraints a network by randomly setting the output of neurons to zero ("dropping out") with a probability p during training. At test time, all connections are still active but their weights are multiplied with p. The effect of dropping out units is that a layer with n units can be interpreted as 2^n smaller networks with parameter sharing that functions as an ensemble classifier. Dropout is a technique that has extensive use in NLP because it increases performance for most tasks. As I will show in chapter 3, dropout is used for almost every single neural network in the RC task.

2.5 Convolutional Networks

This section presents the relevant theory on convolutional networks as they are applied to the task of relation classification. I begin by describing the core theory of convolutions and then move on to how they are applied in NLP.

Convolutional Neural Networks (CNN's) are a relatively novel neural network structure (LECUN REFERENCE) that can be used to process grid-like input such as fixed-length sentences, images, signals or bounded time-series. The central operation in a CNN is the convolution, and then almost always following the convolution is the pooling operation, both of which will be described below. Notice that there is an important difference between the concepts of convolution and a CNN. The term CNN is used to describe any network that contains a convolution operation, while a convolution is a specific linear transformation.

Definitions

I now describe the central operators of convolutional networks, the *convolution*, the *non-linear transformation* and the *pooling* operation.

Convolution operator The first operator in a convolutional network is called a convolution. The convolution is a general mathematical operation defined on two functions f (also called the input function) and g (also called the kernel function) which produces a third function:

$$(f * g)(x) = \int_{-\infty}^{\infty} f(X)w(x - X)dX$$

Since this thesis is about machine learning with neural networks, I will restrict the convolution to be defined on discrete input. This means that we don't need to compute the area under the function of the kernel, but rather can sum the individual values:

$$(f * g)(x) = \sum_{-\infty}^{\infty} f(X)w(x - X)$$

The output of the function $(f * g)$ is called a feature map, or filter map. Correspondingly, when the output is multi-dimensional, a single dimension will be called a feature or a filter.

The above form of the convolution is still not quite relatable to machine learning applications. Usually the input and output of neural network layers are vectors or matrices of real-valued numbers. Text representations can be vectors of words in a sequence, which produces a matrix. To use the convolution in a neural network, I simply define f and g to be functions of the input layer x which produces the input value and the kernel value, respectively. The functions are defined to be zero at any position where the input layer is not defined. If the input of f and g produces real numbers, the input of a network layer will be a vector. The convolution is now defined as a finite summation over the elements of the layer. And finally, I will introduce an intercept term b which can shift the linear transformation away from the origin:

$$S*(x) = (f*g)(x) = \sum_X f(X)*g(x-X)+b$$

I use the naming $S*$ for the convolution because of its application to NLP in the thesis - usually the convolution is over a sentence S. This definition of a convolution will be used in my convolutional networks in the rest of the thesis.

Non-linear transformation Since the convolution is a linear transformation, it is necessary to apply a non-linear transformation in order to be able to approximate non-linear functions in the network(22). Commonly used functions are the sigmoid, tangent or rectified linear unit (ReLU). The non-linear transformation in CNNs is also called the *detector stage*. The non-linearity can be written conveniently on top of the convolution:

$$S*(x) = (f*g)(x) = g(\sum_X f(X)*g(x-X)+b)$$

where g is the activation function.

Pooling operation The last standard operation in a typical layer of a CNN is called the pooling operation. The goal of the pooling operation is to make the output of the convolution *translationally invariant* to small changes in the input. In the context of convolutions as feature detectors, the pooling operation can be thought of as a summarizer of a region of outputs for a certain feature. The operator "pools" over a spatial region of output and summarizes their responses. There are many functions that can be used, including:

- Max-pooling, which simply selects the output from the pooling region with the highest score.
- Average-pooling, which averages all the outputs from the pooling region.
- Weighted average, which weighs the average operation with a metric, usually the distance from the central unit in the pooling region.
- Using the L2 norm of the entire pooling region

When the This rate of reduction is called *downsampling* and can improve the efficiency of the network, since the next layer in the network can have fewer inputs while still detecting important features. Below is shown two examples of max pooling operations:

The first figure shows a layer of convolved outputs where there is no downsampling, and a pooling region of 2. Here, the max pooling have the effect of "overriding" lesser important features with the most important. The second figure shows a global max pooling on the same convolution. Here the intended effect is to select the most important feature of the entire filter. Downsampling and pooling region is set to the entire dimension of the filter. This operation is most useful for single-label classification because it will only select the most important feature.

2.5.1 Motivation for Convolutions

The motivation for convolutional neural networks comes from both biologically inspired theory as well as practical problems with deep feedforward networks.

While it may be the case that the actual design and implementation of a human brain is poorly understood, the idea of applying convolutions in neural networks are undoubtedly inspired from biological principles (18, p. 353). Since the development of CNNs has been a lengthy process, I will only provide a few examples of features of CNNs that are inspired by neuroscience.

1. CNNs are inspired by areas of the brain such as the primary visual cortex (V1). The V1 is arranged as a topological map, which mirrors the input of the eye as it is captured in the retina. One output in convolution can be interpreted as an output from a locally connected neuron in such a map. The neurons on the left and right shares parameters with this neuron. A layer of these neurons forms a kernel in the convolution. By using a CNN to model the V1, it is possible to achieve state-of-the-art predictions on how the V1 responds to visual input(38).

2. The concept of early layers in the model representing simple features or hidden classes of later units in the model is also directly inspired from the concept of simple cells in the human brain (35). Generally, a CNN models simple cells as detector units (the affine transformation). As an example, simple cells are analogous to edge detectors in image object recognition tasks.

3. Similarly, complex cells represent feature detectors that are invariant to small changes in the input. If CNNs are used for recognizing relations or objects in the input, it is advantageous to use a structure that is designed to recognize if some feature is present rather than specifically where. Generally, a CNN models complex cells as the pooling operation which is applied after the detectors. As an example, specific neurons are believed to associate to high-level concepts such as famous individuals (31)

These biologically inspired features provide some clear practical advantages over regular feedforward networks in many practical applications. Traditional feedforward networks are usually fully-connected on each layer. This is problematic when the input grows as the amount of parameters which must be learnt by the network grows linearly. Secondly, when the task of the network is to detect specific features in the input, sharing parameters across the inputs is a way to reduce the number of parameters while still preserving the ability to detect such features. This means that the same entire kernel will be used on all the inputs. As an example, a RGB image object recognition network with 1 hidden layer, prediction of 50 different objects and input image sizes 200×200 will have $200 * 200 * 50 * 3 = 6,000,000$ weights. With CNNs we can design the kernel to be much smaller than the input which greatly reduces parameters while still detecting meaningful features in a practical application. A typical first kernel of a CNN can be as small as $5x5x3$, where 3 is the number of channels in the input image[1]. With parameter sharing, the network will have $5 * 5 * 3 * 50 = 3750$ weights, which is an immense reduction.

2.6 Neural Networks in Natural Language Processing

I now turn to discuss how CNNs can be applied to text processing and specifically sentence-level problems, which include relation classification.

Word Embeddings

In section 2.4 the input layer of a neural network is modeled as neurons that output a real-value scalar or vector. In Natural Language Processing the input for classification

[1]`http://cs231n.github.io/convolutional-networks/`

problems are usually sentences consisting of words. It is not obvious how to convert discrete words in an input sentence to real numbers. A traditional approach is to convert each word into a probability of the word occuring given a previous sequence of $N - 1$ words. This approach is called the *N-gram model* and has been used for many years in NLP (**?** , chapter 4). This approach suffers from many problems, however. The values of an N-gram lack semantic meaning outside of their context and suffer from the curse of dimensionality. A language vocabulary with 100,000 words and $N = 5$ has a potential of $100,000^5 - 1 = 10^{25} - 1$ free parameters, which is huge while 5 words is a rather small window of understanding for a word. A better way of modeling words which also retains their semantic relationship is to use a *distributed representation* for words, also known as *word embeddings*. Each word is represented with a m-dimensional vector. A word embedding can be learned efficiently with a neural network on huge amounts of unlabeled data with modern computers (4). A particular neural network which is used to train word embeddings is called the *skip-gram model*. The idea of skip-gram is similar to n-grams in that it tries to predict words in the context of a sentence. A skip-gram model used to produce word embeddings are also called *word2vec* (25).

Word embeddings are commonly used by training a word embedding on huge amounts of unlabeled data and then subsequently using and fine-tuning the embedding on another NLP task such as RC. When used like this, word embeddings are an example of *representation learning* and *layer-wise pre-training* (**?** , Chapter 15). Another option is to randomly initialize the word embedding layer with random vectors and train them exclusively for a specific task.

In section 2.6 I described how words can be represented as dense vectors which represent semantic information about the words. This semantic information can be transferred from other learned tasks or they may be specific to relation classification (such as positional distance to the entities in the sentence). Assume that we use a vector representation s of size $m \in \mathbb{Z}$ for each word. A sentence S_n consisting of n words $s_1, s_2 \dots s_n$ is then expressed as matrix $M \in \mathbb{R}^{n \times m}$. This matrix can be the target of a convolution. The following considerations and design choices must be taken into account when designing the convolutional kernel for NLP:

- The kernel should have height m so the convolution is over entire words.
- The width of the kernel represent a specific n-gram and is also called the *window size*.
- The depth (also called number of filters) of the kernel determines the number of hidden classes or features that can be detected.
- Differently from convolutions of images, the stride is usually only one. If the stride increases, some n-gram which is important for the classification step may be omitted. In CNNs used for image processing this is alleviated by the pooling strategy.
- Since the kernel convolves over entire word vectors an appropriate pooling strategy must be chosen. Max-pooling can be used and represents selecting the most important n-gram for each filter used in the convolution.
- Finally a padding must be selected for convolution. Choosing a valid padding reduces the number of n-grams which are built only on padding tokens and is therefore considered noisy. A drawback of valid padding is that it reduces the number of convolutions which can be stacked in a deep network. However, convolutions in NLP are usually done in a single layer and so this is not important. A valid padding is usually chosen.

Recent work on relation classification with CNN's follows the above guidelines (**?**)(27)(11). Usually, the network is not deep in the number of convolutions, but it can have several convolutions at the same level to detect different sizes of n-grams. Following the above guidelines and using k filters for each window size w, the output matrix of such a convolution is $S* \in \mathbb{R}^{n-w+1 \times k}$. Since the convolution is almost always followed by a non-linear transformation, I write that transformation directly in $S*$ so that:

$$S*_{i,j} = g(\sum_{x=0}^{w-1} f^T_{x+1,j} f^T_{x+i,j} + b_j)$$

Where b is an applied bias $\in \mathbb{R}$ and g is a non-linear function such as the ReLU, tangent or the sigmoid function. To clarify further, j indexes into the specific filter and i indexes a specific n-gram. Consequently each filter has a vector $s_j = [S*_{1,j}, S*_{2,j} \ldots S*_{n-w+1,j}]$ which represents detecting a specific hidden class from all the n-grams in the sentence. The entire kernel is learnt by the CNN with training. Finally the pooling step is applied to $S*$ to achieve translational invariance and select the most important n-grams for each hidden class. The pooling operation is a global max pooling and produces a vector:

$$P_{S*} = [max\{S*_1\}; max\{S*_2\} \ldots max\{S*_k\}]$$

where $S*_i$ is the i'th filter of the convolution.

Chapter 3

Related Works

Applications of deep neural networks to NLP have been a very active area of research in the past two decades. In relation classification, an abundance of work have gone into improving results on common datasets such as SemEval 2010 and the ACE 2005.

This chapter presents related work in the area of relation extraction and classification. The focus of the chapter is the shift in tendency from heavily hand-crafted *feature-enginereed systems* to neural networks with unsupervised pre-training which require considerably less task-specific feature-engineering. These neural networks are commonly called *feature-learned systems*. Additionally, the chapter also covers approaches that uses supervised learning and approaches which try to use additional unlabeled data during or after the training process.

It is important to notice that the definition *feature-based system* - which is used in many RE and RC papers such as (27), (3) and (26) - is kind of a misnomer. What it usually specifically means is that the features which are used in the learning system are crafted with linguistic knowledge, and that they are not learned by another system. Such features might measure the shape of the word, the words around other words, which specific word group a word belongs to, which part of the sentence the word belong to and so on. Neural networks also use features. These features may not be hand-crafted, or they may be called a representation instead, but they are still features. Word vectors, which are the common input structure used for neural networks in NLP, is both a representation and a set of features for each word which is learnt from an unsupervised task. In neural network terminology, the words features and representation are often used interchangeably. The systems also overlap; for example, (45) combines a word-vector with "traditionally" extracted features as input for a CNN.

3.1 Supervised Approaches

Using labeled data to learn a classifier in a supervised manner is the dominant approach to RC and RE tasks. Since the labeled data is usually raw text (and included in RC, marked entities) some transformed representation must be used in the classifier.

Work on RC and RE are often divided into feature-based and *kernel-based systems* (27) (45). This division can be confusing since *Support Vector Machines* (SVMs) are often used in both types of systems. In common software libraries SVMs always uses a kernel even when treating the input as a linearly separable problem.[1]. For the sake of clarity, I will define kernel-based systems to only include SVMs which uses non-linear kernels and by doing so does not explicitly compute the feature space.

[1]Here are two major frameworks who do just that - LIBSVM (`https://www.csie.ntu.edu.tw/~cjlin/libsvm/`) and SKLearn (`http://scikit-learn.org/stable/modules/svm.html#svm-kernels`)

3.1.1 Feature-based systems

The quintessential model of feature engineered systems in RC is the NLP, which have been extensively used for RE and RC. For these tasks, the input sentence is represented by a series of linguistic features in a vector. The goal of the SVM is to maximize the margin between these training vectors in the dimensionality defined by the vectors.

For the SemEval training set, an extensively cited SVM had the highest accuracy for the 2010 SemEval conference (32). The SVM uses a total of 45 feature types - which amounts to many more actual features when applied to the input. Here is a description of a few selected:

1. *Wordnet Hypernyms*, which are generic classes, or hypernyms, for groups of words, defined by a Princeton research group[2].
2. *Part-Of-Speech Tags*, which describe how each word relates to different parts of speech and grammatical properties.
3. *Dependencies*, which describe how each word are related to the syntactic structure of the tree. Dependencies are analyzed by using a dependency parser on the sentence.

Each individual feature is represented as a boolean value and the feature space is treated as linearly separable.

3.1.2 Kernel-based systems

Because the dimensionality of the feature vectors can be very high, kernel-based systems use the *kernel trick* to implicitly compute the distance in space between the vectors. (8) is a seminal paper on using kernels in NLP. Notice that the input to a kernel may be raw text accompanied by some extracted features, which then forms a new representation. The full feature space is usually an exponential combination of the representation, and it is this space that the kernel avoids. The most commonly kernel-based classifiers are SVMs and the *Voted Perceptron*(14). Essential for all kernels described in the following sections is that they exploit the structural nature of text by either creating parse-trees or sub-sequences that can be analyzed.

Subsequence Kernels

Kernels that exploits the sequential nature of text are called *(sub)sequence kernels*. (6) defines a general subsequence kernel and uses it for the RE task to extract protein interactions from a biomedical corpus and on the ACE 2005 dataset. The used kernel has specific measures for penalizing longer subsequences between relation heads, entity words which are important for RE and RC.

Sequence kernels can also be applied to raw text alone to but is often used in tasks where more text are availaible such as document classification (24).

Tree Kernels

Parsed texts are represented as tree structures which define the relations between words in a sentence. Kernels can be designed that use these structures to compute the vector product. (44), (5), (30) and (28) all use different variants of parse trees in their kernels. The tree-kernel encapsulates a wide array of parsing techniques such as constituent parsing, shortest path dependencies and shallow parsing. Additionally, each kernel is highly specific to the RC task. The kernels are used either in a voted perceptron or a SVM.

[2]`https://wordnet.princeton.edu/`

Feature-Kernel Hybrids

Several systems have augmented the tree-kernel with extensive linguistic features. Such system creates a parse-tree from the input and subsequently annotates each node with a variety of features. Some features which can be added to the tree are:

- *Instance features*, which are added to the root node of the parse-tree. These features can specify information about the marked entities such as their syntactico-semantic structure (7) or their common word type.
- *Phrase features*, which annotates each phrase node with lexical and contextual information(19).
- *Context Information features*, which annotates each node with information relative to the parse tree such as path to the marked to the entities or the context path from a sub-tree root to the annotated node (46)

(37) defines a *feature-enriched tree kernel* for relation extraction which uses all the above defined enhancements to the kernel and achieves (at the time) state-of-the-art performance on the ACE2004 dataset, which is closely related to ACE2005.

3.1.3 Feature-learning systems

Recently, work on the RC and RE task have moved from the above mentioned systems to systems that use less linguistic knowledge as input for the classifier. A sub-goal of these systems is to use an input representation which is as close to the original input as possible. The system should then automatically learn and derive features from the input instead of being input by a human expert. In NLP, these systems are almost exclusively neural networks.

Word Embeddings

An extremely important innovation for neural networks is the word embeddings described in section 2.6. The seminal papers (4) and (25) define the usage for neural networks as a probabilistic language model and the popular word2vec implementation, respectively. Every single modern neural network used for NLP which are covered in this chapter use a distributed vector representation as one of the first layers of the network (40) (11) (26) (27).

Recent work on neural networks in RC can be roughly divided into networks that utilizes recurrent structures and work that uses convolutional kernels as their respective core component.

Convolutional Neural Networks

CNNs have been applied to a wide range of sentence classification and modelling tasks. (9) presented a basic CNN which were used for six different NLP tasks including Semantic Role Labeling which is closely related to RC and RE. The paper defines the basic structure of a CNN for NLP which is the structure used in many subsequent papers:

1. Input sentences are transformed word-by-word to word embeddings.
2. The word embeddings are convolved over with a chosen window size, representing an n-gram selector.
3. The convolved output is pooled and fed into one or more fully connected layers.
4. Finally, the last layer is fed into a softmax classifier designed for the specific task.

This structure modeled the convolution as a linear operation. (23) introduced the non-linearity to the convolution stage and introduced variable-sized downsampling, but is otherwise built directly on top of the previous network. Building on top of recently published CNNs is a noticeable trend in recent years. Specifically for the SemEval 2010 dataset the last five years have been a series of increasing state-of-the-art performances by CNNs (among other networks). (45), (27), (11), (41) and (40) have all expanded on the core idea described above and pushed the accuracy on the dataset. I will briefly describe how each work expands on previous ideas:

-
-
-

Keywords: Domain adaptations Factor-based compositional embedding models

My convnet Relation Extraction: Perspective from Convolutional Neural Networks Nguyen and Grishman, 2015

This conv net is almost exclusively end-to-end, and only uses word vector and positional embeddings which requires no serious feature engineering. Its basically an n-gram selector

The santos net Classifying Relations by Ranking with Convolutional Neural Networks Santos et al. This net is almost like my net but uses a custom trained word vector and

Recursive Neural Networks

The other class of networks which have been used extensively in the past years for NLP and RC is the *recursive neural network* (29). Recursive networks are a generalization of *recurrent networks* (13) which can operate on tree-structures. The idea of a RNN is to learn a function h which for a temporal sequence, of input outputs a state vector and an output for each step. The RNN is a very abstract model for recursive computations, and choosing a suitable h is essential for implementing an effective learning model. The most commonly used model is the *long short term memory network* (LSTM), which computes the state for each timestep by learning a series of memory gates. These gates are learned by the network and serves as controllers that filters input, filters output and updates the state (33). The described networks in this section all use a LSTM or a variant of the original algorithm.

In NLP, a sentence can be seen as a temporal sequence where individual words are timesteps. For RC specifically, sentences are usually parsed and a RNN is applied to the tree to produce an output. (34) did exactly this with the goal of learning composite vectors for phrases and sentences. This work focuses mostly on this representation, which involves creating matrices for every word and learning functions which can combine them. However, the representation were then applied to different end tasks, including the SemEval 2010 task 8, and reported state-of-the-art performance. The above mentioned strategy of using a parse tree as input is key to several recent results for the RC task. Additionally, most of these networks also use other NLP tools such as POS-tagging or WordNet hypernyms. (43) and (12) both uses a *dependency tree* to find the most relevant path of words between entities in a sentence. These paths are then input to a RNN along with linguistic features about each word.

While a sentence is usually read once from left to right in a CNN or a simple RNN, RNNs such as the LSTM can also be modified to read the sentence backwards. A *bidirectional LSTM* (BI-LSTM) records the state at each output as a concatenation of the states achieved by reading the sentence in both directions. (26) uses a BI-LSTM to both extract entities and classify them, and achieves state-of-the-art results on the ACE 2005 dataset.

Their model uses as input both a sequence for the entity detecting and a parse tree for the relation classification.

Finally, RNNs can be stacked to achieve depth in the model. While it is not theoretically well understood why stacked RNNs can perform better than single RNNs, they have shown promising results. (17) mentions several areas of NLP that have shown increases in performance with stacked RNNs. On the SemEval 2010 dataset, (42) uses a deep RNN which achieves state-of-the-art performance albeit with a heavy training cost.

When comparing the RNNs to CNNs, it is important to notice that all of the recent results on RNNs use auxillary input such as parse trees, POS-taggers and WordNet Hypernyms. When these features are excluded, the RNNs in general do not perform significantly better or worse on the SemEval 2010 and the ACE 2005 dataset.

3.2 Semi-supervised Approaches

I'm not sure about this stuff Semi-supervised with clustering:

`https://www.cs.nyu.edu/~asun/pub/ACL11-FinalVersion.pdf`

Chapter 4

Research

As shown in **??**, the research community have come up with a multitude of solutions for the RC problems based on neural networks in the recent years. Models are frequently built upon each other. Techniques such as regularization are done with multiple techniques at an experimental level and the effects may overlap to an unknown extent. Finally, restrictions on computational power and time to execute

4.1 Experiment Details

Try removing L2 constraint if worse results, instead do: early stopping increase dropout

Semi-supervised

Try getting a better accuracy with more data

Reverse augment the data?

On the generalization?

4.2 Model Architecture

Chapter 5

Discussion

Chapter 6

Conclusion

Bibliography

[abu]

[2] Abney, S. (2007). *Semisupervised Learning for Computational Linguistics*. Chapman and Hall/CRC Computer Science and Data Analysis Series 8. CRC Press.

[3] Bach, N. and Badaskar, S. (2007). A review of relation extraction. `http://www.cs.cmu.edu/~nbach/papers/A-survey-on-Relation-Extraction.pdf`.

[4] Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model.

[5] Bunescu, R. C. and Mooney, R. J. (2005). A shortest path dependency kernel for relation extraction.

[6] Bunescu, R. C. and Mooney, R. J. (2006). Subsequence kernels for relation extraction.

[7] Chan, Y. S. and Roth, D. (2011). Exploiting syntactico-semantic structures for relation extraction.

[8] Collins, M. and Duffy, N. (2001). Convolution kernels for natural language. `http://l2r.cs.uiuc.edu/~danr/Teaching/CS598-05/Papers/Collins-kernels.pdf`.

[9] Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning.

[10] Culotta, A., McCallum, A., and Betz, J. (2006). Integrating probabilistic extraction models and data mining to discover relations and patterns in text. `http://cs.iit.edu/~culotta/pubs/culotta06integrating.pdf`.

[11] dos Santos, C. N., Xiang, B., and Zhou, B. (2015). Classifying relations by ranking with convolutional neural networks.

[12] Ebrahimi, J. and Dou, D. (2015). Chain based rnn for relation classification.

[13] Elman, J. L. (1990). Finding structure in time.

[14] Freund, Y. and Schapre, R. E. (1999). Large margin classification using the perceptron algorithm.

[15] Girju, R., Nakov, P., Nastase, V., Szpakowicz, S., Turney, P., and Yuret, D. (2007). Semeval-2007 task 04: Classification of semantic relations between nominals,.

[16] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks.

[17] Goldberg, Y. (2015). A primer on neural network models for natural language processing.

[18] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[19] GuoDong, Z., Jian, S., Jie, Z., and Min, Z. (2007). Exploring various knowledge in relation extraction.

[20] Gurulingappa, H., Rajputc, A. M., Robertsd, A., Flucka, J., Hofmann-Apitiusa, M., and Toldoc, L. (2012). Development of a benchmark corpus to support the automatic extraction of drug-related adverse effects from medical case reports. `http://www.sciencedirect.com/science/article/pii/S1532046412000615`.

[21] Hendrickx, I., Kim, S. N., Kozareva, Z., Nakov, P., Seaghdha, D. O., Pado, S., Pennacchiotti, M., Romano, L., and Szpakowicz, S. (2010). Semeval-2010 task 8: Multi-way classification of semantic relations between pairs of nominals.

[22] Hornik, K. (1991). Approximation capabilities of muitilayer feedforward networks. Neural Networks, Vol.4, Pergammon Press.

[23] Kalchbrenner, N., Greffenstte, E., and Blunsom, P. (2014). A convolutional neural network for modelling sentences.

[24] Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., and Watkins, C. (2002). Text classification using string kernels,.

[25] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality.

[26] Miwa, M. and Bansal, M. (2016). End-to-end relation extraction using lstms on sequences and tree structures.

[27] Nguyen, T. H. and Grishman, R. (2015). Relation extraction: Perspective from convolutional neural networks.

[28] Plank, B. and Moschitti, A. (2013). Embedding semantic similarity in tree kernels for domain adaptation of relation extraction.

[29] Pollack, J. B. (1990). Recursive distributed representations.

[30] Qian, L., Zhou, G., Kong, F., Zhu, Q., and Qian, P. (2008). Exploiting constituent dependencies for tree kernel-based semantic relation extraction.

[31] Quiroga, R. Q., Reddy, L., Kreiman, G., Koch, C., and Fried, I. (2005). Invariant visual representation by single neurons in the human brain. Nature.

[32] Rink, B. and Harabagiu, S. (2011). Utd: Classifying semantic relations by combining lexical and semantic resources.

[33] Schmidhuber, J. (2003). Long short-term memory: 2003 tutorial on lstm recurrent nets.

[34] Socher, R., Huval, B., Manning, C. D., and Ng, A. Y. (2012). Semantic compositionality through recursive matrix-vector spaces.

[35] Srinivas, S., Sarvadevabhatla, R. K., Konda Reddy Mopuri, N. P., Kruthiventi, S. S., and Babu, R. V. (2016). A taxonomy of deep convolutional neural nets for computer vision.

[36] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting.

[37] Sun, L. and Han, X. (2014). A feature-enriched tree kernel for relation extraction.

[38] Vinch, B., Zaharia, A. D., Movshon, J. A., and Simoncelli, E. P. (2012). Efficient and direct estimation of a neural subunit model for sensory coding. Neural Information Processing Systems NIPS.

[39] Walker, C., Strassel, S., Medero, J., and Maeda, K. (2005). Ace 2005 multilingual training corpus. `https://catalog.ldc.upenn.edu/LDC2006T06`.

[40] Wang, L., Cao, Z., de Melo, G., and Liu, Z. (2016). Relation classification via multi-level attention cnns.

[41] Xu, K., Feng, Y., Huang, S., and Zhao, D. (2015a). Semantic relation classification via convolutional neural networks with simple negative sampling.

[42] Xu, Y., Jia, R., Mou, L., Li, G., Chen, Y., Lu, Y., and Jin, Z. (2016). Improved relation classification by deep recurrent neural networks with data augmentation.

[43] Xu, Y., Mou, L., Li, G., Chen, Y., Peng, H., and Jin, Z. (2015b). Classifying relations via long short term memory networks along shortest dependency paths.

[44] Zelenko, D., Aone, C., and Richardella, A. (2003). Kernel methods for relation extraction.

[45] Zeng, D., Liu, K., Lai, S., Zhou, G., and Zhao, J. (2014). Relation classification via convolutional deep neural network.

[46] Zhou, G., Zhang, M., Ji, D. H., and Zhu, Q. M. (2007). Tree kernel-based relation extraction with context-sensitive structured parse tree information.