DIT245 Natural Language Processing
Assignment 3

Jonatan Hellgren

December 2021

1 Introduction

This report will cover the topic of using natural language processing to perform a word sense disambiguation. The goal of word sense disambiguation is to determine the senses of certain words in texts, since the same word can have different meanings depending on the context. For example the word *line* have a different meaning in the sentance: "We waited in line for the ticket sale" and "Your opinions are in line with mine". To see this is rather simple for a humans that are fluent in English but to create a machine learning model for this is a non trivial task.

I will be using the deep learning module `PyTorch` to train the models. I where able to write this code due to the help of the self-study notebooks. Also a youtube video[1] that explained the embedding and lstm layer, it also helped me with a more clear structure in my code.

2 Data

Each instance of the data consists of four parts: the label, the word we are considering, the position of that word and the text with the content. To make the data more accessible for the model a few pre-processing steps where made.

The first pre-processing step we do is to onecold encode all the words, so we will instead of inputting the whole word in to the model instead only input a integer. We will get this integer by counting the frequencies in the train data and using the most common list as a dictionary, so the most common word will be encoded as a one, the second as a two and so on. Here we will limit the model in to a length decided upon with the variable `vocab_size` in the code. All words that are not amongst the most common will be encoded a zero.

The second pre-processing step that will be carried out is to center the words we are trying to disambiguate. We will do this by selecting a value for the window size we want to look at, then we place the target word in the middle of that window. The words previous and following the target word will be placed at the same relative position it had before to the target word. If there exist no word in any position in the window size it will be encoded as a zero.

[1] https://www.youtube.com/watch?v=euwN5DHfLEo

3 Models

To solve this task three deep neural network models where created, all three models are of a similar structure. The first layer in all models is a embedding layer that embeds the encoded words into a certain size which is chosen by the hyper parameter `embedding_dim`. All the out of vocabulary and padding tokens will be placed at the origin in the embedding dimension.

The main difference between the models is what happens after the embedding layer, here each model have different layers that process the embeddings further. All models end with a hidden dense layer that is followed by the final classifying layer.

All models create their own word embeddings, so no transfer learning where used for this task. The model will be trained with a Adam optimizer and a categorical cross entropy loss function.

3.1 Multi Layer Perceptron

The first model implemented used the bag of words method on the word embeddings by summarizing the embeddings over all the tokens.

The sum of the embeddings are then concatenated with a onehot encoding of the target word are being classified. So we are telling the model what word we are classifyin the sense of each time we do a forward pass, but this is after handling the embedding layer. This will help the model not to predict a class that is not linked to the word we are classifying.

With the concatenation from the sum of the embedding layer and the onehot encoding of the word we are classifying, we will use two dense layers with ReLU activation function to get a classification.

3.2 Convolutional Neural Network

With this model the steps are quite similar just that instead of using the bag of words method for processing the word embeddings we will instead use two convolutional layers with ReLU activation function followed by a max pooling layer.

The thought behind the convolution is that the network might be able to find some connections with the words and refine the word embeddings before classifying them.

The pooled layer are concatenated with the onehot encoding of the target word similarly as before. Then results are then fed into two dense layers similarly as the previous model.

3.3 Long Short Term Memory

This model consists of two LSTM layers, one moving from the beginning of the window up to the target word and the second one moving backwards from the end of the window to the target word.

This is inspired by the paper *Word Sense Disambiguation using a Bidirectional LSTM* authored by Kågemark and Salomonsson but is a highly simplified version. The LSTM layers let the model process how the words previous and preceding can affect the sense of the target word.

The states of the hidden neurons in the two layers are concatenated together with the onehot encoding of the target word. The model then follows the same steps as the two previous models.

3.4 Hyper parameters

To make the models easier to compare the models where trained using the same hyper parameters whenever possible. The following hyper parameters where used in all models:

- `vocab_size` = 10000
- `padding_size` = 10
- `n_hidden_units` = 512
- `embedding_dim` = 512

Also with the optimizer the same learning rate 0.001 and l2 regularization 0.001 where used.

In addition to this the convolutional models first layer contained 32 one dimensional convolutions of length 3 and the second one had 64 convolutions with the same length. The pooling layer where a 2d max pooling of size 2.

The size of the hidden states in the LSTM layers where of size `n_hidden_units`.

4 Results

Due to the fact that these models overfit quite fast an early stopping where implemented, where the model that achieved the lowest validation loss will be used when predicting the test data. The models where trained a few times and the results where quite stable, only minor changes in the total accuracy where observed. In Table 1 we can see the accuracy's that the `evaluate.py` script wrote, the dummy classifier is also included for reference. In Figure 1 we can see the training history for all models.

5 Discussion

Compared to the baseline all models achieved a significantly better score. The CNN and LSTM model performed significantly better then the MLP model, however there isn't that much of a difference between te CNN and LSTM model. The reason these two model outperformed the MLP model might be due to the fact that they simply have more parameters or that the things the two more complicated models do actually contributes to the classification some how.

We do not however see such a large boost in performance when applying the more complicated models. This could be that more tinkering with the hyper parameters are required or that we do not have enough data to train these properly. Another possible explanation for this could be that it is the embedding layer and the prepossessing of the data is the main reasons for the performance.

Continuing on that line of thought it would off course be interesting to see how well these models would have performed on an already trained word embedding, and I would have done that if I had more time for this assignment.

Word	Dummy	MLP	CNN	LSTM
All	0.3216	0.7033	0.7218	0.7248
active.a	0.3205	0.7436	0.7436	0.7222
bad.a	0.6086	0.8224	0.8158	0.7961
bring.v	0.2133	0.5734	0.5711	0.5826
build.v	0.2133	0.4656	0.4725	0.4656
case.n	0.2049	0.4780	0.5098	0.5732
common.a	0.2484	0.5817	0.6209	0.6176
critical.a	0.2737	0.6825	0.7299	0.7044
extend.v	0.1814	0.6465	0.6256	0.6395
find.v	0.2315	0.6420	0.6683	0.6874
follow.v	0.1461	0.6174	0.6417	0.6365
force.n	0.1636	0.7826	0.8427	0.8344
hold.v	0.1515	0.6807	0.6058	0.6788
keep.v	0.3937	0.7894	0.7841	0.8021
lead.v	0.1798	0.5056	0.5753	0.5348
life.n	0.2258	0.7285	0.7715	0.7715
line.n	0.8560	0.9496	0.9660	0.9588
major.a	0.3030	0.5795	0.6098	0.6098
national.a	0.2048	0.6941	0.7633	0.7394
order.n	0.2194	0.7920	0.7493	0.7835
physical.a	0.2387	0.6918	0.6767	0.6737
place.n	0.2433	0.7418	0.7715	0.7478
point.n	0.3593	0.7425	0.7754	0.7455
position.n	0.2031	0.6555	0.6941	0.6504
positive.a	0.3585	0.7028	0.7028	0.6981
professional.a	0.2171	0.7771	0.7943	0.8371
regular.a	0.2180	0.6105	0.6424	0.6395
security.n	0.2032	0.8311	0.8602	0.8602
see.v	0.6314	0.7825	0.8114	0.8140
serve.v	0.1553	0.6534	0.6667	0.6780
time.n	0.2796	0.6398	0.7151	0.7204

Table 1: Here we can see the performance of our three models compared to the Dummy classifier beseline

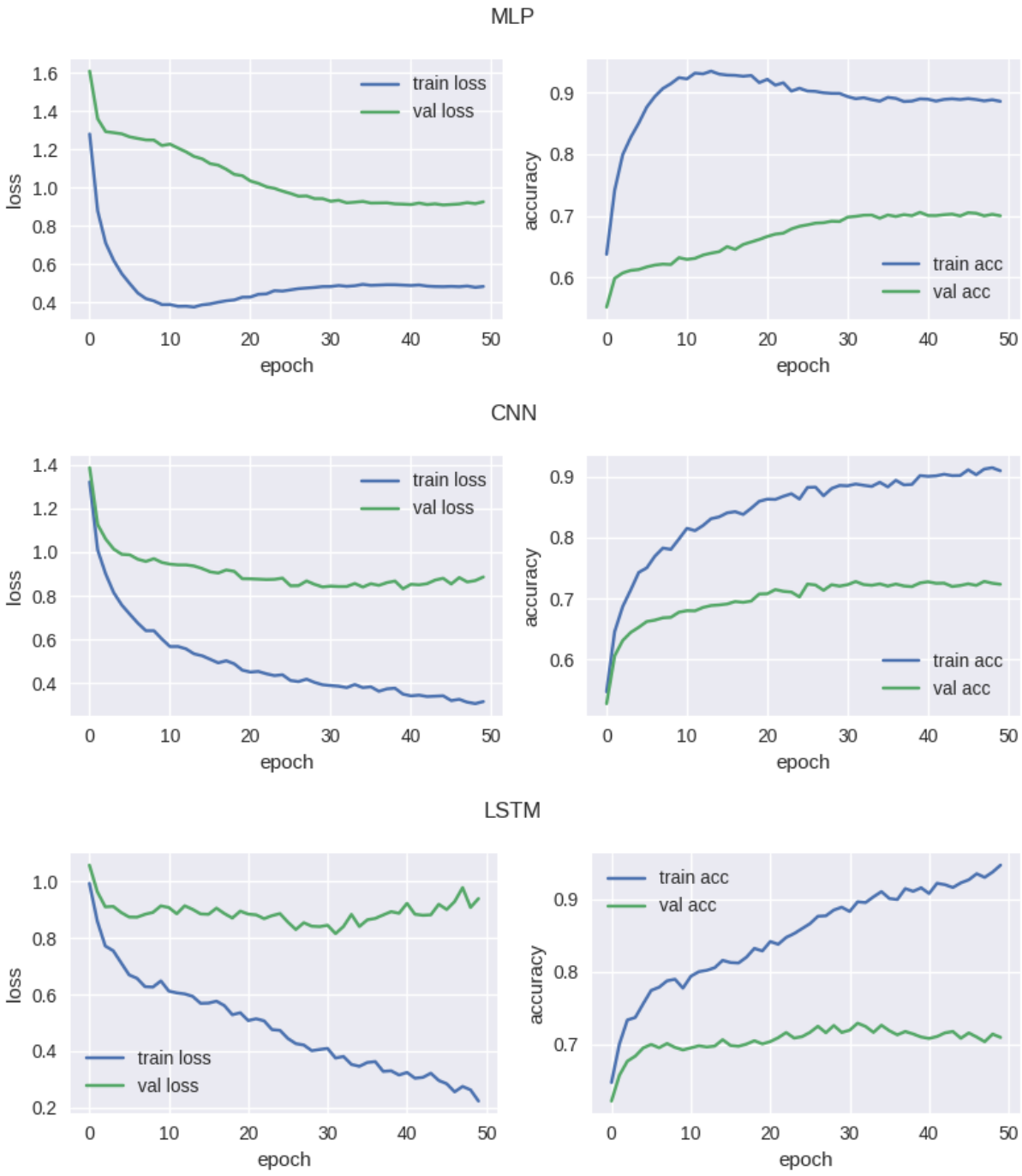

Figure 1: In this figure we can see the training history for each model trained.