IN5550 - Spring 2023

Mandatory assignment 3
Named entity recognition and pre-trained language models

Cornelius Bencsik, Amir Basic & Torstein Forseth
corneb, amirbas & torfor

Git repository: https://github.uio.no/torfor/IN5550/tree/main/oblig2
Fox directory: ../../../projects01/ec30/CocoAmirTorfor

1. Sequence tagging with static word embeddings

For task 1 we have chosen a simple LSTM model to do the classification for us. There is nothing fancy about the architecture, just an embedding layer feeding into a LSTM layer and finally the output layer. The main challenge with this task is to prevent the model from overfitting on the highly over-represented no-entity label “O”. The approach to this task is loosely inspired by our approach that we used in mandatory assignment 2, the only difference so to speak is that we’re working with multiple labels per sentence.

The process from raw text to predictions is as follows:

- Extract useful information, that is sentences and labels from the Conllu file.
- Removing rows from training data that only contains the “O” tag. We decided to remove all sentences from the training data that don’t contain an entity tag because our goal is to predict entity tags, and we believe having a lot of sentences without any entity tags in the training data could create bias towards predicting “O”.
- Text sentences, labels and sentence length is then processed through a dataset class which uses pre-trained word-embeddings to generate the token indexes. Sentences are padded to be the same length using the word-embeddings padding index, and labels are specifically padded with the label “-100”. This is done to make it easier to remove the padded labels and token indexes later on.
- In order to not fit the model to the padded data we have to mask the padded data. In the forward function of the model we are extracting the word index for the padded tokens and then multiplying the padded indexes of each sentence with a very large negative value. This way the gradient will effectively be zero and the model will effectively not learn from these padded tokens.

- After the model is trained we will again evaluate it on a single batch of the validation data. In order to get the F1 score we need to preprocess our outputs as the raw outputs will still have the padded tokens and labels. This is where we can use the value of the labels to remove padded labels and tokens. We simply just remove all labels and their corresponding tokens, where the label value is -100. After the outputs has been preprocessed, we store them in order to calculate the F1 score using the attached scripts for the assignment.
- This procedure yields a F1 score of 0.694 and the process took 85 seconds.

2. Sequence tagging with contextualized language models

Our model takes bert as an argument to create the contextualized embedding layer. The parameters of the bert model are frozen, and the only learnable parameters of the model are in a basic linear classification layer placed on top of the bert model that maps the output from bert to the number of classes we have. We decided to build and use our own linear classifier layer instead of using the BertForSequenceClassification function, even though they basically do the exact same thing.

The process from raw text to predictions is as follows:

- Extract useful information, that is sentences and labels from the Conllu file. We also keep track of the sentence length.
- Splitting data into train and validation sets.
- Removing rows from training data that only contains the “O” tag. We decided to remove all sentences from the training data that don’t contain an entity tag because our goal is to predict entity tags, and we believe having a lot of sentences without any entity tags in the training data could create bias towards predicting “O”.
- Text sentences, labels and sentence length is then processed through a dataset class which uses the bert tokenizer to encode the sentences into tokens, attention marks and offset mappings. The sentences are also used to extract the original length of each sentence before tokenizing. The labels are used to create a label indexer that transforms the original entity tags into numbers representations. This dataset class pads all the data we need so that each element in each separate data type has the same length, so that it can be turned into tensors. The dataset class process the data and gives us all the desired data we need:
    - Input_ids - tokenized sentence
    - Attention_mask - binary representation of tokenized sentence and padding
    - Labels - labels represented as numbers
    - Offset_mapping - character offset of each tokenized word

    - Word_length - length of each word
    - Sentence_length - length of sentence
- Data is split into batches in a data loader
- Model is initialized using the desired bert model and training starts
- During training, before we touch the model, we use input_ids and word_length to create a mapping matrix which helps us map the contextualized embedding output from the subtokens of each word back to the original word.
- We then feed input_ids, attention masks and the mapping matrices into the model. The input_ids and attention masks are fed into the bert model. The output of the bert model is then averaged pooled using the mapping matrix to get a representation of each original word. Lastly, the average pooling representation of each word is passed through the linear classification layer to get our predictions.
- These predictions are then used to compute a loss, computing gradients, and all typical NN training patterns.
- To get predictions after training, we have to perform one more trick since the output predictions are of size [padded_sentence_length, num_labels]. So to get a prediction, we have to take the argmax of the output. Which leaves a prediction of size [padded_sequence_length]. But we want a prediction of size [original_sentence_length], which is where the stored sentence_length comes in; we use this to depad the output predictions, so that the output is of the desired size [original_sentence_length].

The following are the results in terms of F1 and training time of testing different bert models. All the results are trained over 8 epochs and use the same hyperparameters which are learning rate = 0.01 and weight decay = 0.1.

Model	**F1-score**	**Time**
Basic baseline (Task1)	0.694	85s
NorBERT2	0.84	399s
NorBERT3-xs	0.675	359s
NorBERT3-small	0.82	382s
NorBERT3-base	0.85	417s
NorBERT3-large	0.85	595s
Bert-base-case	0.65	464s

2.1 - Fine tuning

Epochs = 8 weight_decay = 0.4

Model	F1-score	Time
NorBERT2	0.84	399s
NorBERT2 - Unfreeze - lr = 0.01	0.00	488s
NorBERT2 - Unfreeze - lr = 0.001	0.00	481s
NorBERT2 - Unfreeze - lr = 0.0001	0.91	473s
NorBERT2 - Unfreeze - lr = 0.00001	0.883	477s

Fine tuning the parameters in the bert model was unstable. Even when we got a great final performance, suddenly some epoch-performance dropped from 0.88 to 0.2 F1 score. Furthermore, as we can see, with a high learning rate, we get 0 correct entity predictions. This is most likely due to the fact that the model instantly learns to only predict “O” since there is a huge overweight of “O”.

Another thing to note is the training time. It is only slightly higher than that of the base NorBERT2 with frozen parameters.

2.2 - Architecture

As mentioned, our base case architecture is just a simple linear layer on top of the BERT model. We also tried out implementing a GRU network and bi-LSTM network on top of the BERT model, with a linear layer at the end for both of them. We used the best performing BERT model from the previous task, that being the NorBERT3-base. We chose the base case over the large model because of training time and model size. Note that learning rate = 0.01 and weight decay = 0.4 since those were the best performing hyperparameters for the NorBERT3-base as seen in the previous part. The motivation for trying a GRU and bi-LSTM layer is simply to see if we can squeeze out performance from the model by adding another complex layer on top. We also try to run it with both 8 and 15 epochs to see if that impacts the performance. Note that the BERT parameters are still frozen.

With NorBERT3-base and frozen parameters:

Model/classifier	**F1**
BertLinear 15 epochs	0.86
BertGRU 8 epochs	0.84
BertLSTM 8 epochs	0.887
BertGRU 15 epochs	0.87
BertLSTM 15 epochs	0.897

We see that the bi-LSTM structure yields the best performance for both 8 and 15 epochs. Meaning that adding a complex layer on top gave positive results and could be worth it. However, we haven't accounted for training time and model size. Also note that just adding a GRU doesn't really improve our results significantly so it seems that if one adds an additional network on top BERT it should be a larger and complex structure such as the bi-LSTM. Otherwise, it's not really worth it.

2.3 - Best model

Our best model is the BertLSTM trained on 15 epochs with lr = 0.0001 and weight decay 0.4 and trainable parameters inside the bert model. The validation F1 score was 0.9365.

Instructions on how to run our best model is in the README.md file