A COMPARATIVE ANALYSIS BETWEEN DEEP LEARNING AND MACHINE LEARNING ALGORITHMS FOR STANCE DETECTION

HAMSA HUSSEIN

Abstract. This paper tackles the shared task of Semeval 2016 task 6 subtask A by using deep learning algorithms. This paper can be treated as continuation of our previous paper, where we tackle the same shared task but use machine learning algorithms. Upon conducting research through deep learning models, we carry comparative analysis between deep learning and machine learning models. Overall, deep learning models outperform machine learning models.

Introduction

The aim of this paper is comparison between Deep Learning (DL) and Machine Learning (ML) models in Stance detection. We have conducted study on ML in our previous coursework where we used Perceptron model, Naive Bayes classification and Support Vector Machine (SVM) across 4 different word embeddings, GloVe, Word2Vec Skip-gram, Word2Vec Continous Bag of Words (CBOW) and TF-IDF. In this paper we carry out experiment on DL models and compare our results with our ML results. The dataset we will be using is the same as our previous paper, stance detection in tweets, Semeval 2016 Task 6 subtask A (refer to the introductory paper done by Mohammed et al. 2016 if needed).

For direct comparison of machine learning algorithms and deep learning algorithms we select three algorithms that semi-directly compare with our machine algorithms we used in our previous paper. The three deep learning methods we use are Feedforward Neural Network (FNN), Convolutional Neural Network (CNN) and Transformer (BERT).

We chose FNN because it directly compares with our Perceptron model, as its a deeper version of it. Its multi-layer version of the perceptron model. For CNN, it is comparable with our SVM model with linear kernel because both can learn hierarchical features. The key difference is that CNN automatically learns feature combinations through filters. For BERT, we expect it to do the best model out of all deep learning models. We will use two types of BERT, full BERT and SBERT and we will carry a comparison between the two.

1. Related Work

Few studies mentioned in our previous paper were done using DL methods, which makes them relevant in this study. We refer to 3 of them, which directly tackle the same shared task we are pursuing.

Wan et al. 2016 uses Google News Word2Vec (pre-trained vectors on part of Google News dataset) as input followed by convolutional layer (with multiple filters), ReLU, max-pooling and a softmax layer. Since Google News Word2Vec is pre-trained on such large corpus, it requires significant memory and storage so we use a better alternative, GloVe. The CNN model outperformed most competitors demonstrating CNN's effectiveness for stance detection. What played a huge role is the versatility of CNN where we can optimize the performance through modifications to the model. In our CNN model, we experiment with such modifications (e.g., changing filter size).

Vijayaraghavan et al. (2016) propose a hybrid CNN architecture combining character-level (CharCNN) and word-level CNNs for stance detection in tweets, achieving a macro F1-score of 0.635. One of the key findings is that CharCNNs excel with sufficient data but under-perform on small datasets without augmentation whereas Word-level CNNs generalize better for targets with limited data. So combining both models improved results by leveraging their complementary strengths.

Zarrella & Marsh (2016) propose an LSTM-based RNN with transfer learning for stance detection, achieving 1st place in SemEval-2016 Task A (macro F1: 67.8). While this may not directly compare with our selected methods, they use Deep Learning method and achieve the highest f1-score. So ultimately we are trying to aim for our model to achieve around the same f1-score of 67.8.

Another study done by Bekamiri et al. (2024) proposes a hybrid framework combining SBERT (a deep learning model) with KNN (a traditional ML algorithm) for patent classification. This study demonstrates that hybrid DL+ML models can outperform pure DL approaches like PatentBERT while offering greater interpretability. We reach a similar conclusion in this current study. We use SBERT + FNN as one of our models.

2. Methods

2.1. GloVe.

In our previous coursework, we came to conclusion that TF-IDF was the best word embedding compared to GloVe and Word2Vec based on our research. But we decided to use GloVe instead because GloVe excels more in Deep Learning. Neural network like CNN and BERT excel at learning hierarchical patterns from dense continuous embeddings. The limitation of TF-IDF is that it feeds the DL models a sparse bag-of-words vectors which lose word order and semantics, forcing DL models to work harder to reconstruct relationships. Also GloVe's pretrained embeddings (trained on Wikipedia/Gigaword) provide head-start semantic knowledge, reducing the data needed.

2.2. Feedforward Neural Network (FNN).

FNN is the simplest type of artificial neural network where information flows in one direction:

$$\text{Input} \rightarrow \text{Hidden layers} \rightarrow \text{Output}$$

The Input layer (x) represents the feature vector of a tweet, e.g., word embedding. We will be using GloVe as our word embedding. Each hidden layer applies a weighted sum followed by a non-linear activation function. For layer l, the output is:

$$h_l = \sigma(W_l h_{l-1} + b_l)$$

Where:

- W_l = weight matrix
- b_l = bias vector
- σ = activation function (we will use ReLU)

For the output later, since we are dealing with classification task, we use softmax activation:

$$y = softmax(W_L h_{L-1} + b_L)$$

The softmax gives a probability distribution over classes (Favor, Against, None). We also have a Loss function, for multi-class classification we use Cross-entropy loss:

$$\mathcal{L} = -\sum_{i=1}^{C} y_i log(\hat{y}_i)$$

Where C is the number of classes, in this case 3, y_i is true label and $\hat{y}_i$ is predicted probability.

This model directly compares with our perceptron model, however we later use Sentence-BERT(SBERT) instead of GloVe to improve the model's accuracy. That specific model directly compares with our later model which is full BERT. The core idea is that SBERT modifies BERT to produce fixed-size sentence embeddings by applying pooling over token-level outputs. So for input sentence $S = [w_1, w_2, \ldots, w_n]$:

- Token embeddings: $T = BERT([CLS] + S + [SEP]) \in \mathbb{R}^{nxd}$
- Mean-pooling: $s = \frac{1}{n}\sum_{i=1}^{n} T_i \in \mathbb{R}^d$

Where d = 384 (default) or 768 (for larger models). We will define terms like BERT, CLS, etc... later in BERT section. So given a SBERT embedding $s \in \mathbb{R}^d$

- Layer 1: $h_1 = ReLU(W_1 s + b_1)$, where $W_1 \in \mathbb{R}^{256xd}$
- Layer 2: $h_2 = ReLU(W_2 h_1 + b_2)$, where $W_2 \in \mathbb{R}^{128256}$
- Output: $o = W_3 h_2 + b_3$, where $W_3 \in \mathbb{R}^{3128}$
- Prediction: $\hat{y} = argmax(softmax(o))$

2.3. **Convolutional Neural (CNN).** CNNs apply filters (kernels) to detect local features in sequential data (e.g., word embeddings of a tweet). Unlike FNNs, CNNs exploit spatial hierarchies via convolutions and pooling. The way CNNs applies filters to detect local patterns in text is similar to how SVM with n-grams captures word combinations. CNN consists of Inputs, Convolution layer, Pooling layer, Fully Connected layer and Loss function.

The input is represented as a matrix, so we let a tweet $X \in \mathbb{R}^{nxd}$, where n is the maximum tweet length and d is the dimension of word embeddings, which is GloVe in our case.

For the convolutional layer, we apply m filters, $W_k \in \mathbb{R}^{hxd}$, where each one has a width (sliding window) of h. For filter k, the feature map $c_k \in \mathbb{R}^{n-h+1}$ is computed as

$$c_{k,i} = \sigma(W_k X_{i:i+h-1} + b_k)$$

Where σ is the ReLU activation function and $X_{i:i+h-1}$ is the window of h words.

For pooling Layer (max-pooling), it downsample each feature map to retain the most salient feature,

$$\tilde{c_k} = max(c_k).$$

It outputs a fixed-size vector $z \in \mathbb{R}^m$, so one per filter.

For fully connected layer, it simply flatten pooled features and pass to a dense layer for classification:

$$y = softmax(W_L z + b_L)$$

Finally, for the Loss Function, we use Cross-entropy loss, same as FNN.

2.4. **Transformer (BERT).** We will be using two types of BERT, Full BERT and Sentence-BERT (SBERT). They both share one foundation, which is tokenization method. They both use WordPiece.

WordPiece Tokenization splits text into subwords (e.g., "unhappiness" $\rightarrow$ ["un", "##happiness"]). It then adds a classification and separator tokens. So given a sentence S = "I love NLP", the WordPiece Tokenization results in

S = ["[CLS]", "I", "love", "NLP", "[SEP]"]

Where [CLS] is the classification token and [SEP] is the separator token (marks end of sentence). After this, SBERT and full BERT differ. In SBERT, we compute token-level embeddings (contextual embedding) through BERT and average all the token embeddings. In full BERT, we use three types of embeddings with one of them being the WordPiece token embedding to feed it to transformer layer.

2.4.1. *SBERT.* For each token, BERT computes:

$$T = BERT([CLS, w_1, \ldots, w_n, SEP]) \in \mathbb{R}^{(n+2)xd}$$

Where

- BERT: A neural network (transformer) that maps tokens to contextual embeddings.
- d: Hidden dimension (768 for base BERT, 384 for distilled models like all-MiniLM-L6-v2).
- T_i: Embedding of the i-th token (each row of T).

After obtaining the token-level embeddings T, SBERT uses mean pooling in which we average all token embeddings.

$$s = \frac{1}{n}\sum_{i=1}^{n} T_i \in \mathbb{R}^d$$

We exclude the classification token ([CLS]) and the separator token ([SEP]). So the dimension for each T_i is nxd. This sentence embedding captures the entire tweet's meaning in a fixed-size vector. We then use this sentence embedding and feed it to the FNN model.

2.4.2. *Full BERT.* Full Bert uses three types of embeddings, token embeddings (WordPiece), position embedding (add positional information) and segment embeddings (distinguish between sentences). It sums all of these and that is the final input embedding that we feed to the transformer layer:

$$E_{input} = E_{token} + E_{position} + E_{segment}$$

The input embeddings E_{input} are passed through BERT's transformer layers. Each layer applies Multi-Head Attention, Residual connection, LayerNorm, and Feedfroward Network (appendix B). At each layer l, the embeddings are transformed, with the initial embedding being E_{input}. After the final layer (12 layers for BERT-base), we obtain the final embedding, T^L. Since we are doing a classification task, BERT uses [CLS] token's final embedding. Do refer to appendix B for more depth on full BERT.

So the key difference between SBERT and full BERT is that SBERT uses all the token embedding (excluding [CLS] and [SEP]) as an output whereas full BERT only uses the classification token [CLS] as an output.

	Test accuracy	F1-score
FNN (baseline)	0.521	0.537
FNN (adjusted weight)	0.513	0.529
FNN (deeper architecture)	0.527	0.542
FNN (word embedding sum)	0.494	0.509
FNN (deeper architecture + word embedding sum)	0.542	0.553
FNN (deeper architecture + word embedding sum + less aggressive adjustments on weights)	0.532	0.541

Table 1. Results for FNN model

3. Experiment

Since the dataset we are using is the same as previous coursework, we used the same line of code to preprocess the data. We changed all texts into lower case, removed mentions, hashtags punctuations and stopwords. We did this for both the training and test data (same as previous coursework). To keep the comparison fair, we used the same dimension as our previous coursework for GloVE, 300. We also used the same line of code to define the function (get_sentence_embedding_GloVe) to obtain the word embeddings.

3.1. FNN.

After obtaining the GloVe embedding, we converted them to pytorch tensors using (PyTorch package). We then created TensorDataset to combine the tensors, one for each train and test. We stored these under the variables train_dataset and test_dataset respectively. We then used DataLoader to split data into 32 batches and shuffled training data.

We then constructed the FNN model. We defined a 2-layer, with the first layer having input of 300 dimension (GloVe) $\rightarrow$ Hidden layer of 128 dimension and the second layer having input of the hidden layer (128D) $\rightarrow$ Output with 3 dimension (Favor, Against, None).

For training we used 20 epochs and each loop we update the model weights using Adam optimizer. We then evaluated the trained model against test data. The results can be seen in table 1. We applied quite few improvements to this model to improve the test accuracy and f1-score:

- Adjusting weights
- Deeper architecture
- Changing mean to sum for better embedding

Adjusting weights is quite straightforward, we simply adjusts the weights accordingly and pass in the argument to the loss function, criterion = nn.CrossEntropyLoss(weight=weights). For deeper architecture, we added one layer, dropout and batch norm. For the last improvement, we changed np.mean() to np.sum() for the word embedding because it preserves magnitude. We also tested a combination of these improvements and recorded the results in table 1. One additional improvement we tried was to be less aggressive on the weights which we also recorded its results in table 1.

	Test accuracy	F1-score
CNN (baseline)	0.567	0.563
FNN (larger filter)	0.565	0.565
FNN (larger filter + adjusted weights)	0.535	0.546
FNN (larger filter + adjusted weights + dropout)	0.558	0.565

Table 2. Results for CNN model

	Test accuracy	F1-score
Full BERT	0.626	0.602

Table 3. Results for full BERT model

3.2. **CNN.** We started with data preparation by defining a text_to_sequence function which converts tweets to fixed-length (50) sequences of 300 D GloVe vectors. We then converted the labels, i.e. Favor, Against, None, to numerical values using label_*map*.

We then defined our CNN model. We apply 100 filters for each size (1-unigrams, 2=bigrams). For filter sizes, we use unigrams and bigrams so that we can have direct comparison with SVM where we used TF-IDF unigrams and bigrams. Our input shape is (batch, 1, 50, 300), so its 1-channel image where $n = 50$ and $d = 300$. For the pooling layer we Max-Pooling which takes the highest value from each filter's output. nn.Linear then combines pooled features to predict 3 classes.

The rest of the method is the same as FNN, we used TensorDataset + DataLoader (batch size = 32), Adam optimizer, 10 epochs and then evaluated the model. Some improvements we include were, adjusting weights, larger filter and adding dropout. All results were recorded in table 2.

3.3. **Full BERT.** We started with converting text to BERT's input format (WordPiece tokens) by using BertTokenizer. We set our model as re-trained BERT model with a classification head for 3 labels (Favor, Against, None).

For data preparation, the BertStanceDataset Class tokenizes tweets with [CLS] and [SEP] tokens and adds padding to max_len=128. It returns token indices (input_ids), real tokens vs padding (attention_ mask) and numerical stance (label). We also used DataLoader but with batch size of 16 because full BERT takes very long time to train and for each training loop we go through each batch, so the lower the batch size, the quicker the model can be trained.

For training, everything is similar to other models, where it feeds to the output variables in the BertStanceDataset class. The main difference is that this time we use AdamW optimizer which is special BERT-friendly optimizer with weight decay. We then evaluate the model and print the test accuracy and f1-score (table 3).

3.4. **SBERT.** We started with using SentenceTransformer ('all-MiniLM-L6-v2') (from sentence_transformer package) which loads pre-trained 'all-MiniLM-L6-v2' model (384D embeddings). We then encoded all the tweets by converting them to fixed-size sentence embeddings. We then normalized the embeddings as its helps with neural network training. We redefined the FNN model as follows:

- FC1: 384 $\rightarrow$ 256 (with BatchNorm and ReLU)

	Test accuracy	F1-score
FNN SBERT (384D)	0.608	0.597
FNN SBERT (768)	0.631	0.630

Table 4. Results for FNN model using SBERT

	Test accuracy	F1-score
Perceptron Model (GloVe)	0.26	0.23
Perceptron Model (TF-IDF)	0.34	0.36
SVM (TF-IDF)	0.59	0.58

Table 5. Results for relevant ML models

- FC2: 256 → 128 (with Dropout)
- FC3: 128 → 3 (output layer for 3 classes)

We used dropout for regularisation and BatchNorm for stable training. For training the model, we used the same optimizer Adam but with learning rate of 0.0005, weight decay and gradient clipping (to prevent exploding gradients). This time we used 15 epochs and evaluated the model, the results are on table 4. We repeated this for SBERT 768D. We simply change the first layer input to 768 rather than 384. The results are also on table 4.

4. Discussion

Since one of the main objectives for this paper is comparison between ML and DL in stance detection, we will sum the relevant results from our previous coursework in table 5. We will first discuss the results we obtained in our experiments for the DL models and then we will proceed with comparison.

4.1. FNN.

If we observe table 1, we can see that all results were somewhat close with each other. Treating the initial FNN model as baseline, we can compare how certain changed lead to improvement in model performance. Firstly, we notice that adjusting weights reduces the test accuracy and f1-score quite a bit. Secondly, deeper architecture in our FNN model improves the model which is something we expected. Thirdly, changing mean to sum in word embedding dropped both accuracy f1-score by roughly 3%, which is quite significant. Our best model is the model with the deeper architecture combined with changing mean to sum for the word embedding.

The reason we tested for combinations is because even though changing mean to sum in word embedding alone resulted in the worst accuracy and f1-score, once we combined it with the deeper architecture, we obtained the best model. So overall, our best model had an accuracy of 54% and f1-score of 55%.

4.2. CNN.

If we observe table 2, we can see that all models are very close with each other. One thing to note is the third model, where we improved the baseline model by adjusting the weights and increasing the size of the filter. Initially, when we just increased the filter size (2nd model), our test accuracy and f1-score pretty much stayed the same (0.2% improvement). However, once we adjusted the weights both the accuracy and f1-score dropped by roughly 3%. We observed similar situation in FNN.

Combining dropout with larger did not improve the baseline model, hence the baseline model is best representation for the CNN, with 56.3% f1-score.

4.3. **Full BERT.** In table 3, we can see that the accuracy of full BERT model is 62% and the f1-score is 60%. This was the most surprising results. We were expecting value that is very near 70%. Few reasons that could explain this are number of epochs and batch size.

For number of epochs we used 3 and for batch size we set it to 16 (compared to 32 in other models). Even with this low number compared to other models, the training of this model took nearly 3 hours with the use of GPU. For each training loop, we go through each batch (16 batches) and then each epoch. So if we set the epochs to 5 and batch size to 32, it would have taken around 10 hours. This is computationally very expensive.

The evaluation code also took roughly 30 mins, so the whole model is expensive to use. We had to take extra precautions by downloading the trained model, in case we end up losing the runtime. This is a huge downside of full BERT. This is one of the reason why we did not try to improve the model.

4.4. **SBERT.** As you can see in table 4, we obtained our best model in this study through SBERT (768) with FNN. Changing the dimension from 384 to 768 improved the model performance significantly.

Since we are also using BERT here, its also computationally expensive but not as much as full BERT. It took around 10 minutes to execute the first model (384D) and 20 minutes to execute the second model (768D).

4.5. **Comparison.**

4.5.1. *FNN vs Perceptron Model.* If we compare the baseline model of FNN with Perceptron (GloVe), we can see that FNN outperforms it significantly, with both its accuracy and f1-score being more than the double of Perceptron model. Also the best Perceptron model has roughly 15% lower accuracy and f1-score than the worst model of FNN. This showcases how having multiple layers really makes a difference in these type of models. So for this comparison, as expected DL outperforms ML.

4.5.2. *CNN vs SVM.* Instead of n-grams, we used unigrams and bigrams in our TF-IDF for SVM (table 5). To keep the comparison fair, we also used filter size of 1 and 2 in our CNN model (baseline, table 2). In both test accuracy and f1-score, SVM was higher roughly by 2-3%. This wasn't expected, but one reason that could explain this results is the use of linear kernel in SVM. Since we are dealing with stance detection, usually the data is near-linearly separable and SVM with linear kernel works well on those type of dataset.

4.5.3. *Full BERT vs SBERT.* Given that SBERT with FNN had better accuracy and f1-score than full BERT as well as being much faster, SBERT is the better option.

5. Future work

One step we could have taken to make this study complete was the use of SBERT in CNN. Our best model was FNN using SBERT, so it would be a fair comparison if we also used SBERT in CNN. Unfortunately, we weren't able to do that due to computational cost of SBERT and full BERT (other executions waited for the full BERT code to finish executing). However, we can expect for the CNN with SBERT to perform a bit better than FNN with SBERT as the baseline CNN was bit better ($\sim 2\%$) than the baseline FNN.

Another step we can take to take this study further is to test a combination of both ML with DL (e.g., SBERT with ML), similar to how Bekamiri et al (2024) did in his study. This study might showcase the best models.

6. conclusion

One key insight that might be significant we noticed is the adjustment of weights. Every time we adjust the weight to counter the class imbalance, we end up with worse performance. However in ML models, adjusting the weights significantly improved the model's performance.

Overall, in general DL models outperformed ML models, which was to be expected. However, we also learned not all DL models are better options compared to ML when it comes this task. A major factor that one should take to account is computational cost when it comes to choosing DL model. Full BERT was the model we expected to perform best, but it did not. While it out performed all the ML, its computational cost was very expensive. So for DL models, a balance is key. Our best model, FNN with SBERT is the right balance, where we utilize BERT and use one of the simplest neural network with it.

References

[1] Mohammad, S. M., Kiritchenko, S., Sobhani, P., Zhu, X., & Cherry, C. (2016). *SemEval-2016 Task 6: Detecting Stance in Tweets*. In Proceedings of SemEval-2016 (pp. 31–41). Association for Computational Linguistics.

[2] Vijayaraghavan, P., Sysoev, I., Vosoughi, S., & Roy, D. (2016). *DeepStance at SemEval-2016 Task 6: Detecting Stance in Tweets Using Character and Word-Level CNNs*. MIT Media Lab, Massachusetts Institute of Technology Cambridge, MA 02139.

[3] Zarrella, G., & Marsh, A. (2016). *MITRE at SemEval-2016 Task 6: Transfer Learning for Stance Detection*. The MITRE Corporation.

[4] Wan, W., Zhang, X., Liu, X., Chen, W., & Wang, T. (2016). *pkudblab at SemEval-2016 Task 6: A specific convolutional neural network system for effective stance detection*. In *Proceedings of SemEval-2016* (pp. 384–388). Association for Computational Linguistics.

[5] Bekamiri, H., Hain, D. S., & Jurowetzki, R. (2024). *PatentSBERTa: A deep NLP based hybrid model for patent distance and classification using augmented SBERT*. Aalborg University Business School, Aalborg, Denmark.

Appendix A. Dataset and Code

This link contains both py and ipynb for my code as well as the both datasets I used, training and testing dataset (same as previous paper).

https://github.com/husseinh6/nlp-assignment-2

Appendix B. Full BERT

Here we will go in depth on how Transformer Layer works in full BERT. So given that X is our input ($X = E_{input}$), each layer applies Multi-Head attention, Residual Connection + LayerNorm and Feedforward network.

For Multi-Head Attention (MHA), its purpose is to capture realtionships between tokens by computing weighted sums of values, where weights are attention scores. The equation it uses is:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d}})V$$

Where Q, K, V are linear projections of X.

The MHA is followed by Residual Connection + Layer Normalization. Its purpose is to stabilize training by mitigating vanishing gradients and normalizing activations.

$$X_{norm} = LayerNorm(X + MHA(X))$$

Where MHA(X) is obtained from previous step.

Then this is followed by Feedforward Network which applies nonlinear transformations to each token independently with ReLU activation:

$$FFN(X_{norm}) = ReLU(XnormW_1 + b_1)W_2 + b_2$$

We then follow it by another Residual + LayerNorm and obtain this output:

$$X_{out} = LayerNorm(X_{norm} + FFN(X_{norm}))$$

The second residual connection preserves information from the attention layer and the final LayerNorm ensures stable gradients for the next layer or classifier.

After going through all 12 layers, we obtain the final output, X_{out} which contains contextual embeddings for all tokens. We then extract the final [CLS] embedding ($X_{out}[0]$) through classification step and pass it through the classifier:

$$Output = softmax(W_c X_{out}[0] + b_c)$$

Coventry University
Email address: husseinh6@uni.coventry.ac.uk