Unveiling the Power of Multi-Layer Feed-Forward Networks in Text Classification

Daniel Pereira da Costa
University of Southern California, Los Angeles, United States
danielp3@usc.edu

Abstract

Text classification is one of the core tasks of Natural Language Processing (NLP). Despite being such a challenging task, reasonable results can be achieved using nonlinear classifiers. This paper discusses the implementation of a multi-layer feed-forward network from scratch, evaluating its efficacy across four different datasets, employing three distinct embeddings. The model's performance is compared across a variety of different configurations, as well as with out-of-the-shelf models.

1 Introduction

The primary goal of this paper is to provide a comprehensive exploration of the methodologies and strategies involved in constructing a Feed-Forward Neural Network. Being able to capture complex underlying patterns, Neural Networks can have many free parameters, making their training a more complicated and time-consuming process compared to simpler models like Logistic Regression and Naive Bayes.

In this paper, we go over the process of implementing a Feed-Foward Neural Network with one and multiple layers, exploring a range of feature and architecture configurations. We assess the impact of using TF-IDF for word selection as the input for a fixed-length feed-forward approach and the influence of momentum on model performance and training time. Additionally, we compare the outcomes of employing word embeddings concatenation versus averaging to represent sentences. All of these analyses are conducted for models with one and multiple hidden layers.

The results demonstrate that simpler models are more suited for this problem and that increased complexity does not invariably lead to improved performance. Furthermore, our study underscores that computing the average of embeddings at the sentence level, rather than concatenating them, has proven to be a more effective approach.

Embedding	File	Dimensions
GloVe	glove.6B.50d.txt	50d
fastText	fasttext.wiki.300d.vec	300d
ufvytar	ufvytar.100d.txt	100d

Table 1: Word Embeddings

2 Methods

2.1 Data Processing

The initial data preparation phase in this article, following techniques from Homework 1, included several key steps. First, contractions were expanded using a dictionary, ensuring standardized text. Next, irrelevant elements like digits, uninformative words, and punctuation were removed to enhance data quality. English stop words were eliminated to prioritize essential information for the machine learning models. Finally, all characters were converted to lowercase, aligning with the lowercase format of embedding files, as discussed in section 2.2.

2.2 Embeddings

In order to increase efficiency, improve generalization, and enhance performance, word embeddings are commonly used in the NLP space, and this paper similarly adopts them as the technique for feature transformation.

As described in Table 1, 3 different embeddings were employed. The first one is GloVe (Pennington et al., 2014), which exclusively comprises English words. The second model is fastText (Bojanowski et al., 2017), encompassing not only English but also a low-resourced language spoken in India. The third embedding model is ufvytar, which contains words from an unidentified language.

A <UNK> token was incorporated into each embedding file to address Out-of-Vocabulary (OOV) instances. This token is associated with an embedding vector derived from the mean of all the vectors present in the embedding file.

Matrix	Dimensions
X_{input}	$[n_{doc}, max\_length \cdot emb_{dim}]$
W_1	$[max\_length \cdot emb_{dim}, h_{units}]$
b_1	$[h_{units}]$
W_2	$[h_{units}, num\_labels]$
b_2	$[num\_labels]$
A_1, Z_1	$[n_{doc}, h_{units}]$
A_2, Z_2	$[n_{doc}, num\_labels]$

Table 2: Matrices shapes

3 FeedForward Network

3.1 Input Layer

Since the embeddings are the word level, we need first to find a way to apply them to each of the sentences of variable length. We set a maximum word limit for each sentence to address this, as determined by the parameter *max_length*. In cases where a sentence contains fewer words than the specified *max_length*, we pad the vector with zeros at the end. By using this technique, the input matrix (X_{input}) will end up with the shape described in Table 2, where n_{doc} is the number of documents, and emb_{dim} is the embedding dimension used.

3.2 Forward Pass

Two weight matrices are created as we construct a one-layer feedforward neural network. Specifically, W_1 facilitates the transition from the input layer to the hidden layer, while W_2 manages the transition from the hidden layer to the output layer. Correspondingly, we employ bias vectors b_1 and b_2 to complement these weight matrices. All of the matrixes were randomly initialized.

For each epoch, we first compute the output of the input-to-hidden layer by $Z_1 = X \cdot W_1 + b_1$. Next, we apply the ReLU activation function to obtain $A_1 = ReLU(Z_1)$. Moving on to the hidden-to-output layer, we once again employ a similar formula: $Z_2 = X \cdot W_2 + b_2$. However, given that this is a multiclass problem, we apply the softmax activation function, $A_2 = softmax(Z_2)$. The dimensions of all of these can be found on Table 2.

3.3 Backpropagation

In this step, we must compute all gradients to update the weights and biases matrices described in the previous section. By applying the chain rule, it is possible to derive all of the gradients from the equations described in Section 3.2 and obtain the equations in Table 3. Once the gradients are computed, we can update the weights: $W_i = W_i - l_r * dW_i$ and $b_i = b_i - l_r * db_i$ where l_r is the learning rate and i = [1, 2]

Matrix	Dimensions
dZ_2	$A_2 - y\_mini\_batch$
dW_2	$1/m_{minibatch}A_1.T \cdot dZ_2$
db_2	$1/m_{minibatch}np.sum(dZ_2, axis = 0)$
dZ_1	$dZ_2 \cdot W_2.T \cdot ReLU'(Z_1)$
dW_1	$1/m_{minibatch} \cdot X_{input}.T \cdot dZ_1$
db_1	$1/m_{minibatch}np.sum(Z_1, axis = 0)$

Table 3: Backpropagation gradients

4 Extra Mile

TF-IDF: Rather than utilizing the initial *max_length* words from each sentence, as detailed in Section 3.1, an alternative approach has been implemented. This approach involves ranking the words according to their TF-IDF scores. Using this approach, only the words with the highest TF-IDF scores would be passed to the input. The underlying concept is to extract the most significant words from each sentence, considering their importance within a document relative to the entire corpus of documents.

Optimizer: To accelerate convergence, a momentum coefficient was introduced into the stochastic gradient descent algorithm. This was achieved by preserving the weight increment from the previous step and subsequently adding it to the next step after multiplication by the momentum coefficient γ that takes values between 0 and 1.

Average Embedding: The average embedding technique consisted of computing the average of all words embeddings within the max_length limit instead of concatenating them. Consequently, the input dimension changes from $max\_length \cdot embedding\_dim$ to only $embedding\_dim$.

Multi-Layer: In order to increase the flexibility of the neural network, the capability of adding N layers was also included in the from-scratch implementation. Each matrix is now stored in a list, allowing for easy reference in the subsequent hidden layers. Furthermore, the dimensions of the weight matrices now depend on the specific layer. Thus, we can represent them as $W[L_{i-1}, L_i]$, where L_{i-1} denotes the number of units in the preceding layer, and L_i represents the number of units in the current layer. The equations used in both the forward and backward passes are similar, but we must consistently reference the appropriate previous and subsequent layers in the multi-layer approach.

5 Experiment

5.1 Data

Each model was evaluated across four datasets: questions.train.txt, producs.train.txt, odiya.train.txt, and 4dim.train.txt. As validation sets were not provided, the training files were partitioned into 80% portion for training and 20% for validation.

Each dataset was associated with a specific embedding file used to extract features. For products.train.txt and 4dim.train.txt, which contain English words, we utilized the glove.6B.50d.txt embedding. For odiya.train.txt, we employed the fasttext.wiki.300d.vec embedding. Lastly, for questions.train.txt, we utilized the "ufvytar.100d.txt" embedding.

5.2 Model

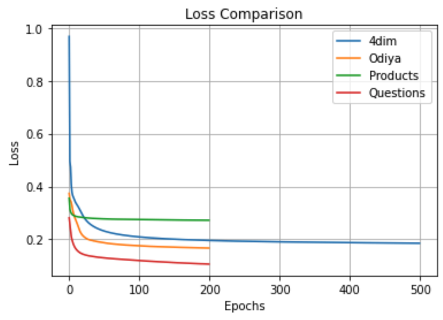

Figure 1: Best MLP from-scratch implementation - Cross-Entropy Loss

A manual hyperparameter tuning process was performed for every model listed in Table 4 to enhance performance. This involved fine-tuning several key hyperparameters, including max_seq_length ([10,20,30, 50]), mini_batch_size ([32, 64, 128, 256]), learning_rate ([0.0001:0.3]), and the number of epochs ([100:500]). The tuning process was executed to minimize the loss of each model, with early stopping employed to halt the training when the loss reached a convergence point, as demonstrated in Fig. 1.

An evaluation was also performed using a combination of different methods that were described in previous sections. This involved adjusting the neural network's hidden layers and units, incorporating TF-IDF for word selection, and optimizing the momentum coefficient.

This evaluation was also performed using a combination of different methods that were described in previous sections. This includes varying the number of hidden layers ([1,2,3]) and their hidden units ([5:100]). It also includes using TF-IDF to select the words used in the model and the momentum coefficient. Furthermore, the model's performance was assessed by comparing it to a PyTorch implementation with a similar architecture as the "Original" model.

5.3 Results

From Table 4, we can observe that TF-IDF alone did not provide an increase in performance for odiya, questions, and products datasets. However, for 4dim there was a slight gain in performance when using this method. This can be attributed to the dataset's smaller size, consisting of only 1,560 rows, where input modifications can more noticeably impact the output.

By incorporating momentum, we again observed a higher gain in performance in the 4dim dataset when compared to the "Original" model. However, we observed a decrease in training time. Specifically, when using a momentum value of 0.4 for the "Products" dataset, the model achieved convergence 17.7% faster.

It is noticeable that the "Average Embedding" method notably enhanced the model's performance. When comparing this implementation to the "Original," "TF-IDF," and "TF-IDF + Momentum" approaches, the "Average Embedding" method consistently outperformed them in 7 out of the 8 configurations. Surprisingly, this method led to an astonishing 59% improvement in performance for the 4dim dataset when compared to the "Original" model.

When comparing the best from-scratch implementation with PyTorch, we can observe that they achieved similar results, with an average difference of only 3.56% across all datasets. Curiously, the PyTorch implementation has not demonstrated a reduction in training time for the same architecture, occasionally exceeding the time required by the from-scratch implementation. In the case of the 4dim dataset, the from-scratch implementation takes 4s to train, whereas the PyTorch is 6.38s. This unexpected result can be attributed, in part, to feature pre-processing conducted in regular Python, which can be slower compared to optimized tensor operations utilized by PyTorch.

When comparing the multi-layer model with the single-layer model, it becomes evident that the accuracy difference is generally insignificant. The 2 and 3 Hidden Layer performed best in

Model	Method	Accuracy (%)			
		odiya	questions	4dim	products
1 Hidden Layer	Original	75.89	65.77	40.71	62.10
	TF-IDF	74.84	45.60	45.51	66.23
	TF-IDF + Momentum	75.66	42.67	42.67	66.17
	TF-IDF + Momentum + Avg Embedding	78.52	67.60	**64.74**	71.00
	PyTorch	80.89	**75.55**	64.40	70.19
2 Hidden Layers	Original	77.20	62.71	28.53	67.33
	TF-IDF	50.27	30.07	29.81	59.01
	TF-IDF + Momentum	75.20	31.05	28.85	59.01
	TF-IDF + Momentum + Avg Embedding	79.87	47.43	57.37	**71.22**
	PyTorch	**82.01**	57.46	21.47	66.40
3 Hidden Layers	Original	46.78	44.50	53.53	70.39
	PyTorch	81.51	44.25	30.13	59.01

Table 4: Performance evaluation results of models on four datasets with test dataset. "Original" denotes the model without additional feature transformations and optimization. "PyTorch" signifies the PyTorch implementation of the "Original" model, utilizing a similar architecture.

Model	Accuracy (%)							
	odiya		questions		4dim		products	
	Test	Bilnd	Test	Bilnd	Test	Bilnd	Test	Bilnd
MLP	79.87	78.3	67.60	73.2	64.74	62.5	71.22	70.6
PyTorch	82.01	81.1	75.55	85.8	64.40	70.0	70.19	70.9
NB (BoW)	91.97	92.18	68.70	74.0	81.85	95.0	77.88	83.4
LogReg (TF-IDF)	84.38	82.37	73.35	71.6	80.20	80.0	70.83	79.4

Table 5: Performance evaluation results of best MLP and PyTorch models on the local test set and Vocareum's blind test set.

"odiya" (15,200 rows) and "products" (32,592 rows) datasets, whereas the single layer outperformed in the "4dim" (1,560 rows) and "questions" (5,452) datasets. These outcomes align with the observation that training a multi-layer model demands a larger dataset due to its increased complexity.

In terms of the models' generalizability, we can notice from Table 5 that the models did not exhibit significant performance degradation on the "odia" and "products" blind tests, with accuracy scores varying within the range of 0.71 to 1.0. Interestingly, some models even achieved a higher performance in the "questions" and "4dim" blind test. These results reinforce the notion that the models were not overfitted and demonstrate their ability to generalize. In cases where there was a decrease in accuracy, such as the from-scratch model on the "4dim" dataset, this may be attributed to the presence of out-of-vocabulary words in the blind test set.

In a comparison between these models and the models from Homework 1, it is noticeable that, on average, the HW 1 models outperformed the newly implemented ones. The HW1 models achieved an average accuracy of 86.15% for Naive Bayes (Bag of Words) and 78.3% for Logistic Regression (TF-IDF). In contrast, the from-scratch implementation achieved 71.2%, and PyTorch achieved 76.95%. Consequently, this implies that less complex models were the more suitable choice for these datasets. The TF-IDF and BoW of words methods used in HW1 proved to be particularly effective for this problem. Additionally, the sensitivity of neural networks' performance to their hyperparameters suggests that they may still operate with suboptimal settings, requiring more tuning.

6 Conclusion

The from-scratch model that achieved the best average accuracy across all datasets was the 1 Hidden Layer (TF-IDF + Momentum + Avg Embedding) with an average of 71.15%, whereas for PyTorch was the 1 Hidden Layer implementation with an average of 76.95%. This study also leads to the conclusion that the feature transformation of computing the average of word embeddings has demonstrated greater efficiency compared to the concatenation method. It enhanced the models' performance and led to shorter training times. Furthermore, it has been observed that simple count-based features combined with less complex models like Naive Bayes and Logistic Regression proved to be more effective for addressing this classification problem. Increasing model complexity by adding more hidden layers has also been found to be ineffective, taking into account the costs and overall performance of the models. Future work could involve testing alternative neural architectures, such as CNNs and RNNs, and comparing their performance to the implementations conducted in this study.

External Sources: Grammarly and ChatGPT (link) were used as spell-checker tools.

References

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.