Bag of Words Document Classification using Feedforward Neural Network and Recurrent Neural Network

JESPER SØBERG and MAGNUS TYSDAL*†, University of Stavanger, Norway

In this project two neural networks were implemented for classifying article abstracts to their scientific fields. A Feedforward neural network (FFNN) and a Recurrent neural network (RNN) were implemented and trained to give their predictions based on a Bag of Words given as input. After training and testing, both network types were able to reach good results, such as an F1-score in the range of 0.7-0.8. It was also discovered that some parameters had a large impact on the network's predictive capabilities, such as high learning rates, a large amount of hidden layers for FFNN, and a large size for the hidden state of the RNN. Ultimately, the networks had similar performance with FFNNs having a tendency for larger computing times whereas RNNs suffered from a risk of exploding gradients.

ACM Reference Format:
Jesper Søberg and Magnus Tysdal. 2024. Bag of Words Document Classification using Feedforward Neural Network and Recurrent Neural Network. 1, 1 (April 2024), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

As part of our chosen project "Bag of Words Document Classification", we were provided with a dataset containing abstracts from various scientific papers along with a label corresponding to the field within which the paper belongs. The primary objective is to train a neural network to accurately predict the correct label of a given abstract.

Achieving this goal involves a number of steps: Every abstract needs to be converted into a format which can be understood by a network. The network itself will naturally have to be implemented and trained on the provided training data. After training, the provided test data will be employed to evaluate how well the network is performing. The following sections will further elaborate on the details of how each of these steps have been implemented.

Thereafter, the later sections will discuss and compare the results that have been obtained by tuning parameters and testing different models and solutions.

2 Preprocessing

The dataset given consisted of two csv files containing the training and the testing datasets. Each data entry has an ID, the contents of the abstract and a label signifying the appropriate scientific field. There are a total of ten different labels found within the dataset:

- Electrical Engineering and Systems Science (eess)
- Quantum physics (quant-ph)
- Physics (physcis)
- Statistics (stat)
- Math (math)
- Astrophysics (astro-ph)
- Condsened Matter (cond-mat)
- High Energy Physics - Theory (hep-th)
- Computer Science (cs)
- High Energy Physics - Phenomenology (hep-ph)

Leaving the labels as is would present a problem however. Neural networks do ultimately only operate on numbers. In order to evaluate (and train) a network's performance, a method is therefore needed to translate each label into numbers. We opted to define each label as a 1x10 vector consisting of 0s and a single 1. Each position corresponds to a certain label, and whichever position contains a 1 indicates which label the vector in question is meant to represent. As an example, the following two vectors correspond to the labels "eess" and "quant-ph" respectively.

$$[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$
$$[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$$

Such a representation of the labels enables a model to judge whether its predictions are correct or not.

Upon reading the dataset, a specified number of entries are selected to be included in a pandas dataframe, dictated by a parameter we call nrows. This is the first of many hyperparameters that will be used to adjust model performance. The aforementioned dataframe is run through some simple preprocessing such as removing punctuation and converting all characters to lowercase before being passed to other functions such as the one responsible for converting labels to vectors as was just discussed.

3 Bag of Words (BoW)

Just as the raw labels are meaningless to a neural network, the abstracts will also have to be converted to some numerical representation for a network to function. Bag of words is a text model that numerically represents the words in a text, such as the scientific papers of our case. The words are placed in a "bag" in the sense that there is no ordering of the words as in the original text. Precisely how the words are numerically represented depends on the specific method used to create the bag. For this project, we use two different methods: A simple count vector and term frequency-inverse document frequency (or TF-IDF for short).

3.1 Count Vector

To obtain a count vector representation of a given abstract, we simply count how many times each term occurs in said abstract.

*Both authors contributed equally to this project.
†All code used can be found at: https://github.com/JesperSoberg/dat550-prosjekt

Authors' address: Jesper Søberg; Magnus Tysdal, University of Stavanger, Stavanger, Norway.

ACM XXXX-XXXX/2024/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The result is a list of all the unique terms in the abstract text along with a number, signifying how many times said term appears within the abstract. Hence we have a vector of numbers which can then be fed into our neural network.

3.2 Term Frequency–Inverse Document Frequency (TF-IDF)

A count vector is appealing due to its simplicity, but does have some shortcomings. We therefore make use of TF-IDF as well, which measures the importance of a term in a document rather than simply counting occurrences. This measurement is made up of two parts:

- Term Frequency (TF): The measurement of how frequent a word is for a specific document. Simply counting occurrences means that longer documents will have higher average occurrences compared to shorter documents. Calculating the frequency of a word instead avoids this issue by accounting for the length of the original text.
- Inverse Document Frequency (IDF) is a score of how rare a word is across all other documents present in the data. The English word "the" for example will usually have a high frequency. But because the frequency is so high in all documents, the IDF score will be low unless a document uses the term significantly more often than other documents. Conversely, terms with a relatively low frequency may nevertheless be seen as important due to rarely being used in other documents.

Thus TF-IDF is simply a value that signifies the importance of a term for a specific document in relation to all the other documents in the dataset. A higher TF-IDF value shows that a specific term is significant and unique to a certain document while a lower value shows a lower importance and that the term is common across all documents.

4 Neural Networks

Neural networks are models that make decisions using processes that are inspired by the layers of biological neurons in a human brain. Each instance of a layer, be it the input, hidden or output layer, are called a neuron or node. Different neurons will "activate" depending on the problem in such a manner that they should be able to weigh the different alternatives possible and arrive at a correct conclusion. For our case here, it would be to read the abstracts and guess its field label given a set of alternatives it can take. In practice this is a matrix with various values which through a number of matrix operations help the network predict a correct answer. When an input is given to a neural network, it will do what is called "forwarding", which is simply going through the network's hidden layers and having the matrices of the hidden layers applied to the input matrix which in turn will results in some output. During training, this output is compared against the correct label such that an optimisation algorithm can adjust the parameters of the network in order to obtain a better result next time.

Before elaborating on each model in the following sections, we will here discuss some traits common to the architecture of both models. First of all is the output of the network: We explained previously that all labels are replaced with a vector of 0s and a single 1. A perfect network should also provide such vectors. The output layer therefore contains 10 neurons, each corresponding to a position within the output vector. The values of these neurons also have a softmax function applied to them to create a probability distribution, which can be interpreted as the network saying how likely it believes each label to be. This is important during the training phase, as it allows the optimisation algorithm to judge how close a network is to the correct answer, and therefore how much it needs to correct the network in order to achieve better results.

Then there is the matter of the input to the network: Each node in the input layer corresponds to a single term within the vocabulary of whichever BoW representation is being used. The size of the input layer is therefore dynamically decided upon first creating the network. Said vocabulary is created when reading the training data, and its size will therefore typically be in the upper 3000s. It is then important to store this vocabulary, as all training the network goes through is based upon it. As an example, consider the network in the following figure:

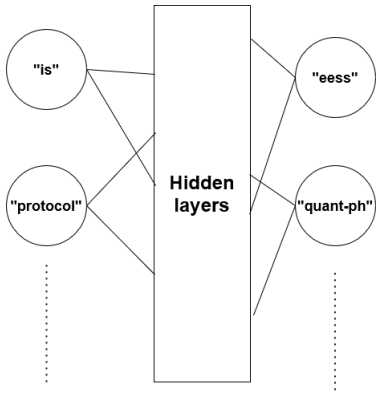

Fig. 1. A network trained on a vocabulary where "is" and "protocol" appear as the first two terms.

Although the network is unaware of what term each number in the input vector actually corresponds to, it will through training learn how it should treat the "first" or "second" term in the input ("is" and "protocol" in this case). If we then were to give it another input using a different vocabulary where the first term is "park" as an example, then the network would treat the word "park" as though it had the same meanings and implications as the word "is". This is the reason why we store the vocabulary created upon first training the network, and always reuse that same vocabulary on any test data later provided. Doing so also means that any terms in the test data that do not appear in the training data will be ignored by the network.

In this project the feedforward neural network and the recurrent neural network have been implemented using the "PyTorch" Python library.

4.1 Feedforward Neural Network (FFNN)

A FeedForward Neural Network (FFNN) is the simplest form of a neural network and generally what has been described in preceding

sections. It consists of layers of nodes interconnected with each other, and uses those connections to derive its final output.

4.1.1 Implementation

Upon creating an instance of the FFNN, it takes a few arguments which are needed to establish its dimensions:

- **Vocabulary size**: As discussed above, the network needs to know how many terms are defined in the vocabulary so that it can create a node within its input layer for each term in the vocabulary.
- **Number of hidden layers**: The number of layers between the input and output layer is entirely up to the user and therefore taken as an argument here

In defining the FFNN implementation, we have it define its input layer first and foremost. The size of this layer is dependent upon the "size_vocabulary" which is again dependent on the nrows we decide before running. Then the hidden layers are created, which is determined by the "num_hidden_size" variable we select before running. It goes through a loop, iterating and creating a hidden layer up to the "num_hidden_layers" value. A final output layer is then created with a size of 10, where 10 corresponds to the number of field labels the neural network can choose between. The hidden layers in-between the input and output all have a size that is equal to the average of the input and output size. This was a rather arbitrary choice, but has overall lead to good results as will be discussed in later sections.

Defining the forward pass then is quite simple: Any data passed into the network is passed through the network layers, with a ReLu activation function applied after each layer to achieve non-linearity. This continues until the output layer, where a softmax function is applied as discussed before.

4.2 Recurrent Neural Network (RNN)

Recurrent Neural Networks are similar to FFNNs in that they have an input, hidden and output layer. However, RNNs are recurrent as the name suggests, meaning they loop over the same hidden layer multiple times before eventually deciding on an output.

4.2.1 Implementation

There were three main parameters we were interested in when defining the characteristics of the RNN implementation:

- **Vocabulary size**: Defines the input size just as in the previous section about FFNN's vocabulary size.
- **Hidden size**: Determines the size of the "hidden state" of the RNN layer. A larger hidden size should allow it to learn more features of the input data.
- **Number of layers**: How many RNNs are stacked on top of each other. Helps in providing a higher learning capacity, given a larger dataset compared to what is required by a single layer.

As the input size is dependent on the vocabulary (which is dependent on number of rows read) this is not adjusted directly as this will change when its dependencies change.

The hidden size and the number of RNN layers are parameters we are able to change freely as they do not depend on anything.

To keep the networks as comparable as possible we use ReLU as the activation function for the RNN as well.

Forwarding

When forwarding through the RNN, we start by initialising the hidden state of the RNN layer as a matrix of zeroes. The input matrix is then reshaped from being a two dimensional matrix to a three dimensional one. This is because the RNN layer takes an input of three dimensions:

- The number of data points, as defined by how much training/test data we forwarding through the network.
- The number of inputs within each data point, which in our case corresponds to the number of terms in the vocabulary.
- The number of features within each input. An input could in theory have many features, but is only a single number in our case (the TF-IDF score or count vector representation). Hence the reason we add a third dimension of size 1 to our input.

The final output of the network is obtained by first retrieving the final hidden state of the RNN layer and passing that through a fully connected layer. Just as before, this fully connected layer consists of 10 neurons and will have a softmax function applied to it before returning the result. The listing below shows the forwarding function of the RNN for the sake of clarity:

```
def forward(self, x):
    h0 = torch.zeros(
    self.numLayers, 1, self.hiddenSize).to(x.device)

    x = x.reshape(x.shape[0], 1,  x.shape[1])
    out, _ = self.rnn(x, h0)

    out = out[:, -1, :]
    out = self.fc(out)
    out = self.softmax(out)

    return out
```

Listing 1. The forward function of the RNN

As seen in listing 1, h0 is the initial hidden layer state. X is the input data that is being reshaped to accommodate for what the RNN expects to receive. Where "x.shape[0]" is the number of rows in the data sent in, "1" is the number of features, and x.shape[1] is the vocabulary.

5 Training & Testing

When conducting the training and testing there were some prerequisites that had to be in place before we could start with the training and testing loops:

- **An optimisation algorithm** had to be chosen. We chose to use Adaptive Moment Estimation (Adam) as it seemed to be a popular default choice.
- **A loss function** was also a requirement to conduct training and testing. Since our project revolves around multi-class classification we chose the Cross-Entropy Loss (CLE) as it is commonly used for such problems.
- **Batching** was also used in order to introduce some randomness into how the training/test data is utilised during training/testing. We implemented this by creating a custom data set class from PyTorch. The class is filled with pairs of data: The first half being the BoW representation of an abstract, and the second half being the binary 1x10 label vectors mentioned in section 2.

  This custom dataset is then passed to an instance of the PyTorch "DataLoader" class. Looping over this DataLoader will return a batch of random samples from the dataset per iteration, which we then use to train or test the network.

We have structured the code such that it will loop for a specified number of epochs; within each epoch, it will first train the network on the training data before using the test data to evaluate its performance. This is repeated for each epoch in the hopes that the network performance will increase with each iteration.

5.1 Training

The training function loops over the DataLoader and passes each batch of data into the forward pass of the network in an attempt to predict the correct labels. Said predictions are then passed to the loss function alongside the true labels in order to calculate the loss. Once this calculation is done, the optimisation algorithm is used to update the weights of the network. Finally, once the loop through the DataLoader is finished, the average training loss is calculated and logged to Weights & Biases (which we will discuss further later).

5.2 Testing

The general structure of the test function is similar to that of the training function. It differs in that rather than updating any weights, it computes the different measurements that we are interested in to measure the overall performance of the network. The measurements that we compute, in addition to loss, are as follows:

- **Accuracy**: A ratio of the number of correct predictions against the total number of predictions.
- **Precision**: Measures how many positive predictions actually are true positives.
- **Recall**: Compares the false negatives against the true positives, by division.
- **F1-score**: A harmonic mean between recall and precision. Hence this measurement acts as a way of balancing between the two.

Since this is a multiclass classification problem, we calculate the macro average of precision, recall, and the F1-score. This means that these statistics are first computed for each individual class, before the average score of each class is returned as the final value.

All of these measurements, as well as the loss calculated during testing, are also logged to Weights & Biases at the end of the function.

6 Hyperparameter Sweeps

There is an overwhelming number of hyperparameters to tune when training neural networks and no easy method of predicting their optimal values. We therefore used Weights & Biases (W&B) to perform a hyperparameter sweep. This involves defining a set of parameters and a function to run with those parameters within a configuration which is in turn uploaded to a central controller at W&B. We can then deploy agents within our own machines to run the function over and over again. The controller will select values for each hyperparameter and distribute jobs to run to each agent.

The "function" referred to here consists of the code described throughout sections 2-5. Each run will log its test metrics to W&B making it easy to compare the performance of different networks using different parameters and thus determining what the optimal set of parameters are for achieving the best score. There was one issue however: W&B did not provide a way of indicating dependencies between hyperparameters. As an example, the hyperparameter "hidden_size" is only meaningful when running an RNN, as the FFNN has no hidden state. Running a sweep with the network-type and hidden_size as hyperparameters would therefore not only waste resources, but would also mislead W&B into thinking the hidden_size is less important than it truly is (as half of the runs would indicate this parameter has no effect on the result).

We ultimately circumvented this issue by creating two separate sweep-configurations: One which always runs an FFNN, and one for the RNN. Both of which had the following hyperparameters in common:

- **nrows:** The number of rows to read from the training data, thus increasing the amount of data available for training.
- **batch_size:** How many data points to include in each batch retrieved from the DataLoader. A smaller batch_size would mean more iterations per epoch.
- **bow:** Which BoW implementation to use; either count vector or TF-IDF.
- **learning_rate:** The learning rate employed by the adam optimiser.
- **num_epochs:** For how many epochs to run the train/test loop.

Futhermore, there were a few hyperparameters specific to each network model:

- **num_hidden_layers:** Exclusive to FFNN and specifies how many hidden layers the model should contain.
- **num_layers:** Exclusive to RNN and specifies how many RNN units should be stacked on top of each other; otherwise known as a layer.
- **hidden_size:** Exclusive to RNN and specifies the number of features in the hidden state of the RNN.

Finally, the controller needs a goal to base its comparisons on. Out of all our metrics, we decided to maximise the F1-score. F1-score was preferred over recall or precision as it essentially is the mean of the two, and our use case does not provide any reason to favour one

of these metrics over the other. The choice between F1-score and accuracy was more arbitrary in nature, but ultimately insignificant as a network performing well in one of these metrics should also perform well in the other. Minimising the loss was also an option, but we decided to use the F1-score instead as we would rather have the model be correct if not so confident as opposed to being overly confident at the cost of more incorrect answers.

7 Results

In the following, we will first detail our major findings after running the hyperparameter sweeps. We will then discuss some smaller experiments we conducted and their results before explaining how all these results were applied to train our final models, and naturally the results from those models as well.

7.1 Hyperparameter Sweep Results

FFNN

The sweep for the FFNN was eventually stopped after finishing 342 runs. Its results are summarised in the diagram below:

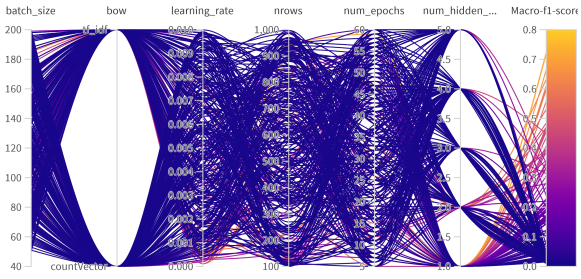

Fig. 2. Results of the FFNN hyperparameter sweep

The diagram shows each individual run as a line moving through the diagram. Following these lines shows which parameters are chosen by said run and the F1-score they eventually resulted in. Unfortunately, most runs in the diagram tend towards a very low score. Although that does make it all the more important to find out what those that do well are doing to differentiate themselves from the rest. W&B provides another diagram which is useful for this purpose:

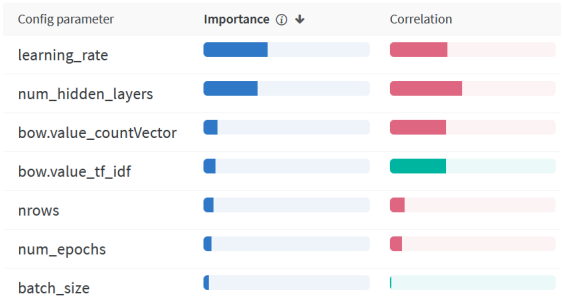

Fig. 3. Analysis of FFNN hyperparameters

Within the diagram are all of the hyperparameters ordered by their "importance". Their correlation to the final score is also shown on the right, where a red colour indicates a negative correlation whilst green indicates a positive one. "Importance" in this case refers to how useful a given hyperparameter is for predicting the final score of a run. The most notable results from this diagram are the negative impacts of a high learning rate and high number of hidden layers, both of which have a negative correlation with the score (of -0.347 and -0.437 respectively). Furthermore, the BoW method being TF-IDF rather than count vector also has a noteworthy correlation with the results (of 0.338) although its importance is not ranked as highly.

RNN

The RNN sweep was able to finish with a total of 809 runs. More than twice as many runs in other words; achieved in half of the runtime no less. Just as the FFNN, the corresponding RNN results are shown below:

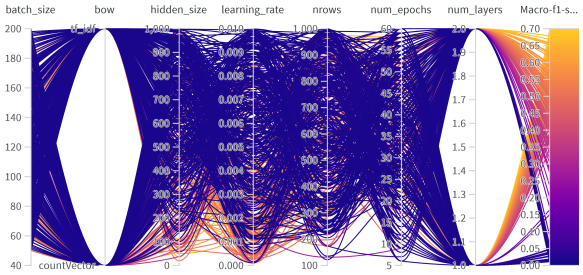

Fig. 4. Results of the RNN hyperparameter sweep

Compared to the FFNN, the RNN generally performed much better. On the other hand, the best RNNs never managed to reach the same heights as the best FFNNs despite the superior number of runs. The RNN hyperparameter analysis is shown below:

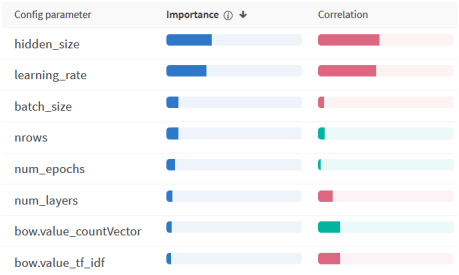

Fig. 5. Analysis of RNN hyperparameters

Overall, the results are very reminiscent of those from the FFNN. It reiterates the importance of a low learning rate and has also found an increase in hidden size to be detrimental to the overall performance of the network. (Their correlations were respectively -0.430 and -0.451.) Increasing the number of layers is also once again found to not be very helpful. Perhaps the most intriguing result here is that although the BoW method used is ultimately insignificant, the sweep appears to favour runs that use the count vector rather than TF-IDF (although the correlation is only 0.164 this time).

7.2 Loss Functions & Optimisers

Other smaller experiments with different parameters have also been conducted which were ultimately left out of the larger sweeps. These involved trying a different loss function and a different optimiser. We had a look at these options:

- **Loss function:**
  - **Cross-Entropy Loss**, the recommended loss function to use for a multi class classification problem like ours.
  - **Kullback-Leibler Divergence Loss (KLDiv)**, a loss function often used for generative models. We chose to further examine it as it appeared to be similar to CEL and also commonly applied to multiclass classification.
- **Optimiser:** In addition to the previously mentioned Adam algorithm, we also wanted to experiment with the Stochastic Gradient Descent (SGD) algorithm due to it being another common choice.

These were tested in a few much smaller sweeps and logged onto W&B to see the results of their performance. Below is a diagram showing how well they were able to maximise the F1-score. All other parameters have been omitted from the plot for the sake of clarity. (nrows is only included to help separate the different lines in the diagram from each other.) Do note that the parameters have been randomly chosen by W&B during these runs.

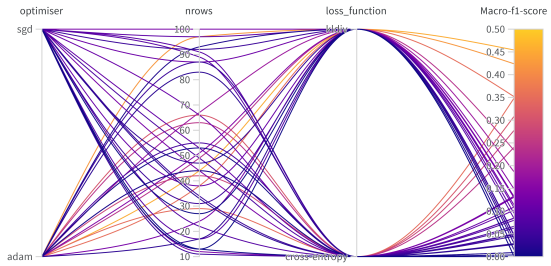

Fig. 6. Performance of different loss functions and optimisers

Here we are able to see that SGD consistently underperforms, whereas Adam is used by all higher-ranking runs in the diagram. Looking at the loss functions we observe that KLDiv has a trend of bad performance but still manages to achieve the highest performing runs whilst CEL often lands somewhere in the middle. Given these results, we decided it was not worth considering SGD as a viable candidate for the larger sweeps as it would most likely give a bad performance and waste computing time when running the sweeps. KLDiv was more challenging to directly compare against CLE as it returned negative loss values. Although it did show some promise, CLE was already well-integrated within our code and this promise was not enough for us to change that.

7.3 Training our Final Models

After obtaining the above results, and a deeper understanding of our parameters, it was time to train our final models. In order to do so, we looked at our top-performing runs from each sweep and designed models using those as a basis. The top 5 best-performing runs from our sweeps have been highlighted in the figures below:

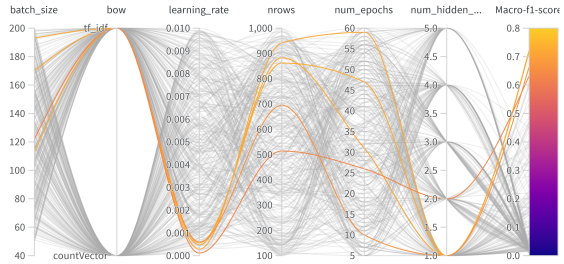

Fig. 7. Top 5 best-performing runs of the FFNN-sweep

These results mimic the ones derived from the sweep as a whole: The BoW method should be TF-IDF, the learning rate should be low, and the number of hidden layers should be low. batch_size, nrows, and num_epochs all appear to be much more flexible.

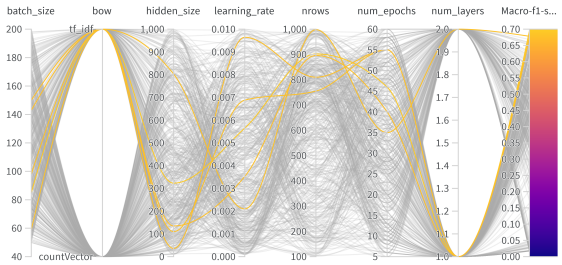

Fig. 8. Top 5 best-performing runs of the RNN-sweep

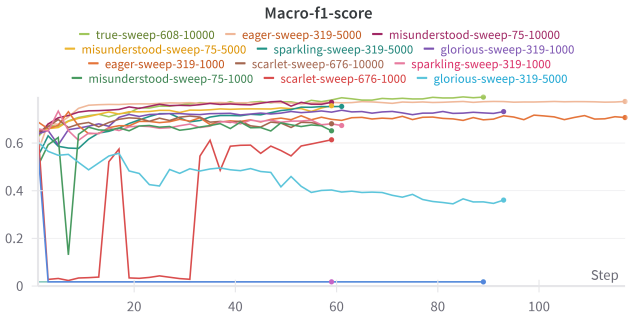

Fig. 9. F1-score of our final runs

The top-performing RNNs show somewhat more interesting results. The number of layers once again favours lower values and hidden_size generally tends toward lower values (although it clearly is capable of performing well with higher values). nrows and num_epochs are further restricted to higher values compared to the FFNN, but most interesting is the rather large variance in learning rate this time. It's also noteworthy that despite our sweep results suggesting the RNN performs better with a count vector, all of the top performers use TF-IDF.

With all of this in mind, we set up an environment locally similar to the function used during the sweep. Within this environment, we define our models and their hyperparameters in a json file and store the weights of the networks after training so that they can easily be recreated later. We decided to copy over the parameters from our 3 best performing networks of each network type, and made some adjustments such as restricting them all to using TF-IDF or only using 1 layer. The exact parameters used can be seen in the code of our GitHub repository.

Finally, a note on the parameter nrows: The results we have shown so far have suggested that this parameter is of little import to the performance of the network, which may seem counter intuitive as one would expect a network with more training data available to showcase better performance. This may be due to an oversight when we set up these experiments, as the parameter nrows not only dictates the breadth of training data available, but also the breadth of the test data. Hence a better trained network will also have a harder test to pass after its training is finished, which may give the impression that increasing nrows does not lead to a significant increase in performance. Thus we decided to run these training experiments with an increasing number of nrows (1000, 5000, and 10 000), in hopes of reaching the highest performance without placing too large a strain on the computer.

7.4 Final Results

In summary, our final runs involved running the 3 highest performing runs from each sweep with some minor adjustments to their configuration for a total of 6 runs. Furthermore, we wanted to gradually increase the number of rows from 1000 to 5000 to 10 000 for a grand total of 18 runs; All of them named after the original run that inspired them, and a suffix indicating the number of rows being used. Their F1-scores are all shown in the following figure:

There are a number of things to note here, but we will begin with the eye-catching performance of true-sweep-608-1000, true-sweep-608-5000, and scarlet-sweep-676-5000, all of which having converged to a score close to 0. We shall discuss this further in the discussion section, but we note for now that they all are RNNs and as we shall see, these runs have all landed on a test loss of NaN.

Another thing to note is that there are only 15 runs in the figure, rather than 18. The 3 missing runs are all FFNNs configured to use 10 000 rows of data, and therefore crashed during execution. The one outlying run hovering around a score of 0.4 is also an FFNN. This one is a bit interesting so keep its light-blue colour in mind as we will return to it shortly.

But with edge cases out of the way, most runs have indeed been successfully recreated and with a score within the 0.6-0.8 range.

The plots for our other metrics are shown below, but now rescaled to ignore the outliers and better highlight the differences between the top performers:

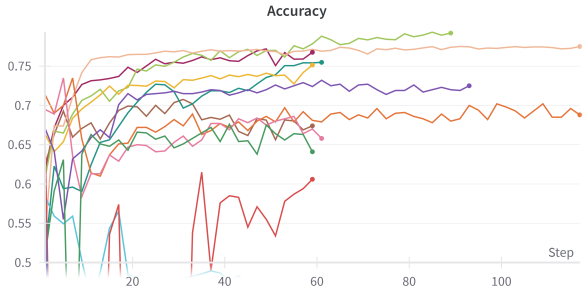

Fig. 10. Accuracy of our final runs

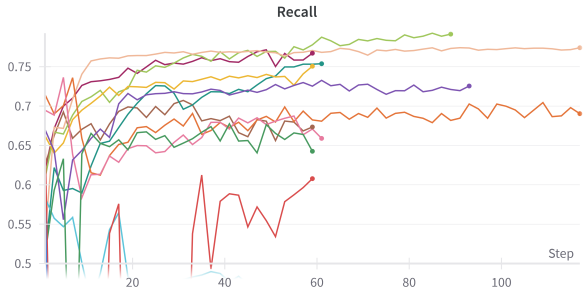

Fig. 11. Recall of our final runs

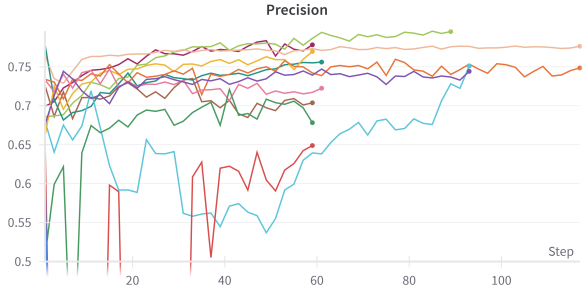

Fig. 12. Precision of our final runs

These plots help clarify why our light-blue run from earlier appeared to underperform. Its recall is just as bad as the F1-score shown earlier, but its precision is actually quite high. Unfortunately, the F1-score is calculated as a harmonic mean of recall and precision, meaning it favours the lower of the two metrics, and this low recall therefore results in a low F1-score.

Finally, the following two plots show the training and test loss for our final runs, where the dotted lines at the bottom indicate a loss of NaN:

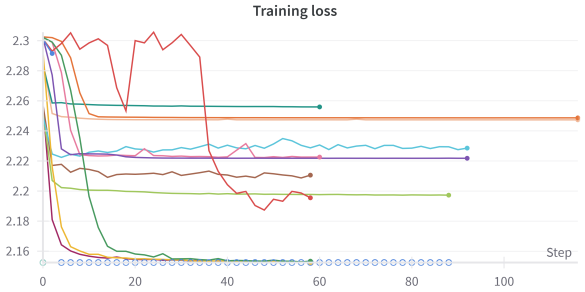

Fig. 13. Training loss of our final runs

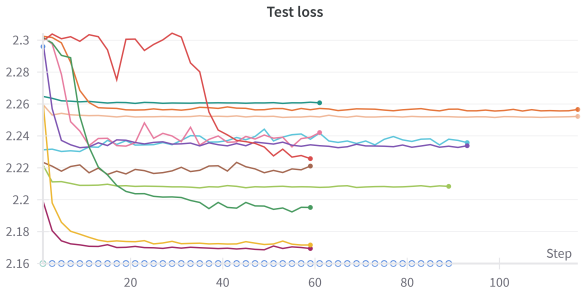

Fig. 14. Test loss of our final runs

After successfully reproducing these networks, the final test was to verify that these networks had indeed managed to learn something from the training data, rather than simply being absurdly lucky. We therefore chose to rerun some of them on other sets of test data to verify that they were able to maintain their good results. Below are the results obtained from rerunning misunderstood-sweep-75-1000 on different test sets. The added suffix indicates the size of the test set.

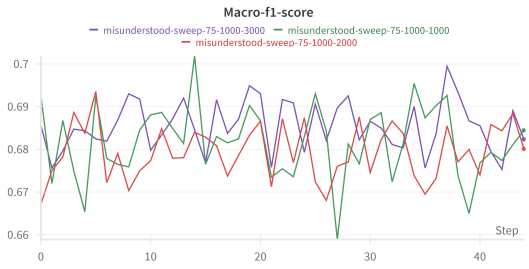

Fig. 15. Rerunning misunderstood-sweep-75-1000 on different sets of test data

These numbers are consistent with those from the original run of misunderstood-sweep-75-1000.

8 Discussion

Finally, there are a few topics from throughout this project which we would like to discuss more thoroughly here now that we have obtained our results.

8.1 Larger ngrams

There was an attempt to implement a larger ngram size for the vocabulary being passed to the BoW models. Throughout the project, the size of the ngrams have been one word per ngram, which potentially leaves out key terms that are distinct to a specific document. As an example, the two word term "machine learning" can be split into the individual words "machine" and "learning", both of which can characterise a plethora of different fields. The combined term "machine learning" on the other hand can be a great indicator of what field an article abstract belongs to. Having a vocabulary of this nature should give the neural networks a boost in performance as they could catch distinct multi-worded terms in the abstracts.

Sadly, the idea was not highly prioritised as the sklearn library used to create the count vector and TF-IDF matrices was not compatible with the implementation of larger ngrams. This could of course be worked around, but implementing and fine-tuning the neural networks themselves took priority over implementing larger ngrams and this idea was ultimately forgotten by the time of experimentation.

8.2 Network Type

It has been evident throughout this project that the FFNN takes significantly more time to train compared to its RNN counterpart, despite RNNs having a reputation for taking longer to train due to their sequential execution. This could be caused by the fully connected nature of the FFNN. Having every node of a given layer connected to every node of the following layer leads to a large number of weights that all need to be adjusted during training. In contrast, the RNN only features a single connection to a single input from each node within the hidden layer. Combined with the fact that the size of its hidden state has been relatively small throughout our experiments, this has likely lead to the much shorter runtime. We expected that these small hidden_sizes would result in an underperforming network, but our results have shown the RNN generally performing better than the FFNN whilst also requiring significantly less runtime and being capable of handling larger amounts of data.

It is also noteworthy that although the RNN may be more consistent than the FFNN, no RNN was able to significantly outperform the best FFNNs as one might have expected. We believe this is due to our solution of feeding the network a Bag of Words in a sequence defined by the vocabulary of the network. Doing so means that one of the main advantages of an RNN - that being the ability to process data sequentially - is somewhat lost as there is no meaningful information in the sequence of the vocabulary. In an abstract on the other hand, the sequence of words is very important as the precise meaning of a word depends on its surrounding context. This sentiment reflects what was just discussed regarding longer ngrams.

8.3 Network Parameters

The number of hidden layers for FFNN had a significant impact on its performance, where almost all runs with more than one hidden layer were unable to reach an F1-score above 0.4, with the sole exception of a single four-layer-run achieving a score of 0.4486. This goes to show that adding several layers is not only computationally costly, but also gives no guarantee of better results as all the best-performing networks had a lower amount of network layers.

Similarly in the case of RNNs: The number of RNN units stacked on top of each other (determined by the num_layers parameter) shows that while it may perform well with two RNN layers, it does not seem to be as fruitful as having just one RNN layer for similar, if not better, performance at a lower computational cost.

RNNs' hidden layer size also had an interesting result, as the previous figure 5 from the results section shows a high negative correlation with large hidden state sizes. One would expect that the larger the hidden state, the more patterns it would be able to learn, and thus its performance would improve. We believe this parameter combined with the learning_rate is the cause behind certain RNN runs reaching a loss value of NaN. This issue appeared in some of our final runs, but has indeed been present during the sweep runs as well, and indeed only for RNNs. We believe the core issue here to be that the RNNs are experiencing exploding gradients. Given that the vocabulary is typically a few thousand terms long, the unrolled RNN effectively has thousands of hidden layers. When this is combined with large hidden states or learning rates, the risk of gradients exploding also grows quite large.

8.4 Loss function

Looking back at our results now, it does appear as though the KLDiv loss function might have been worth exploring further. Although the sample size was very small, and the results therefore inconclusive, KLDiv did achieve higher scores than what those using CEL were capable of. Continuing to experiment with this and other loss functions could very well provide another way of further optimising the performance of the network.

9 Conclusion

In this project, we have identified some important parameters and key challenges in training two different network models to perform a multiclass classification task. One of our best performing models was an RNN which managed to reach 0.79 with its F1-score. Our experiments have shown that both FFNNs and RNNs have little to gain from mindlessly adding more layers. They also suggest that TF-IDF indeed provides more useful information to these networks than a count vector representation. To summarise our comparison of the FFNN and RNN models, we generally found both network types to be capable of similar results. We did however encounter issues with the computational complexity of FFNNs growing faster than that of RNNs. RNNs on the other hand, did have an unfortunate tendency to encounter exploding gradients, especially for larger learning rates and/or sizes of their hidden state.