Document Classification with Feedforward and Recurrent Neural Networks

Data Mining and Deep Learning (DAT550)

Ali Hussain Khan, Said Vagap
University of Stavanger, Norway

ABSTRACT

This project aims to develop and evaluate different neural network models to classify scientific paper abstracts into their respective fields. A subset of the Arxiv-10[1] dataset is used, containing abstracts and labels across ten fields of science. The project employees three methodologies to determine the best outcome.
The first method employs feed forward neural networks with bag-of-words features (CountVectorization and TF-IDF). The second method introduces GloVe[2] pretrained embeddings with three types of pooling (max, mean and sum) into neural networks. The final method explores recurrent neural networks, experimenting with different architectures (Simple RNN, LSTM, GRU, Bidirectional RNN).
Model evaluation is based on training, validation, and testing accuracy to determine the most efficient and well-generalized approach.

1 INTRODUCTION

This project aims to develop and explore various neural network architectures to classify scientific research paper abstracts to the relative fields based on their text features. We use a subset of the Arxiv-10[1] dataset which consists of scientific paper abstracts and their respective labels. The dataset is divided in two sets training and testing.
First step consists of data exploration and preprocessing on the dataset to make it suitable for training use, preprocessing is done on the both sets to ensure consistency.
In the second step we do text vectorization, both CountVectorization and TF-IDF vectorization. Then its neural network development and tuning phase. Two models developed and tuned to fit different type of vectorization.
Third step moves onto use pretrained embeddings. We use Glove[2] for our pretrained embeddings. We aggregate word embeddings from the tokens with three different pooling methods. Using max, mean and sum pooling methods give us a chance to experiment and determine which one performs the best. Three neural network models tuned for each type of pooling for best performance.
Fourth step brings RNN into the experimentation. We use base preprocessed dataset for this section again. Developing four different types of RNN enables performance evaluation. We develop and tune Simple RNN, LSTM, GRU, Bi RNN in this phase of the project. We give training, val and testing accuracy high importance as the goal is the achieve the best all round model, a model which is efficient, simple, high performing and well generalized. We compare all models from all three steps, and find the best performing model.

Project in Data Mining and Deep Learning (DAT550), IDE, UiS
2025.

2 BACKGROUND

In this project, we work on **document classification** for scientific abstracts. The goal is to automatically predict the research field of a paper, such as mathematics, physics, or computer science, based only on its abstract. This is a **multiclass text classification problem**, where each abstract is assigned one label from several possible fields.

Simply put, the computer learns to read and understand scientific texts and then decides which subject, such as mathematics, physics, or computer science, the article belongs to.

Text classification is an important part of Natural Language Processing (NLP) because it helps organize and understand large collections of documents. However, it also brings challenges like handling many different words, technical vocabulary, and understanding the meaning of word sequences.

To solve this, we use different machine learning methods:

- **Bag-of-Words (BoW)** with a simple **Multi-Layer Perceptron (MLP)** to create a basic model by counting word frequencies.
- **Pre-trained word embeddings** (Word2Vec, GloVe, FastText) to give the model a better understanding of word meanings.
- **Recurrent Neural Networks (RNNs)**, including **Simple RNN**,**LSTM**, **GRU** and **Bidirectional RNN**, which are better at handling sequences of words.

We compare these methods using accuracy, precision, recall, and F1-score to find out which one works best for classifying scientific abstracts.

3 DATASET AND PREPROCESSING

The dataset used for this project is a subset of the Arxiv-10[1] dataset, consisting of around 80,000 scientific abstracts written in English. Each abstract is linked to a specific research field, such as computer science, physics, or mathematics. In total, there are 10 different research categories.

The data is provided in a compressed CSV file format, where each row includes:

- **abstract**: A short summary of the paper (input text).
- **label**: The corresponding research field (target label).

Since this is a **multiclass text classification** problem, the goal is to predict the correct research field based only on the abstract.

3.1 Data Exploration

Before preprocessing, we explored the dataset to better understand its structure and characteristics.

3.1.1 Label Distribution. We analyzed the distribution of research fields. As shown in Figure 1, the dataset is well-balanced, with each label having roughly the same number of samples. A balanced dataset is important because it helps prevent the model from favoring one category over others.

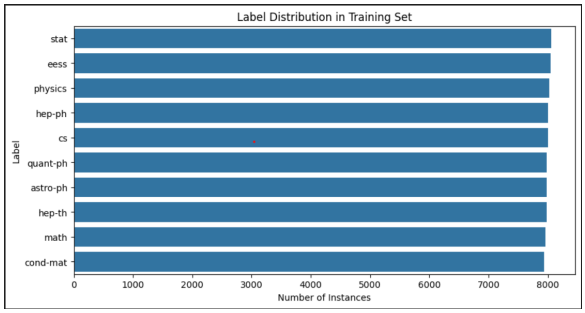

Figure 1: Label distribution in the training set. Each research field is equally represented.

3.1.2 Abstract Length Distribution. We also looked at the length of the abstracts, measured by word count. As shown in Figure 13, most abstracts contain between 100 and 200 words, with an average length of about 150 words. This information helps in deciding an appropriate input size for the models.

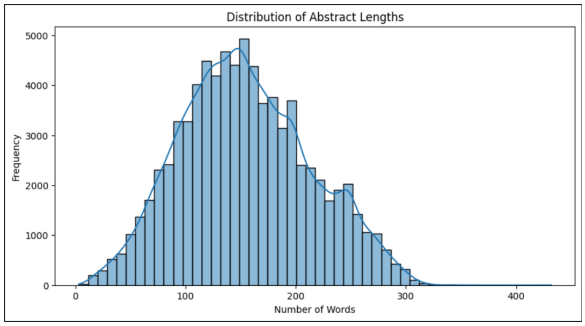

Figure 2: Distribution of abstract lengths in the training set. Most abstracts fall between 100 and 200 words.

3.2 Preprocessing Steps

Before building the models, it was important to properly prepare the dataset. Although the data was already quite clean, a few essential preprocessing steps were necessary to make the text ready for machine learning.
Before explaining each step in detail, here is a brief overview of the preprocessing process:

- **Handling Missing Values**: We confirmed there were no missing entries, so no additional handling was necessary.
- **Text Cleaning**: We removed extra spaces, trimmed leading and trailing spaces, and converted all text to lowercase. This ensures that words like “Physics” and “physics” are treated the same.
- **Tokenization**: Abstracts were split into individual words (tokens) using regular expressions.
- **Label Encoding**: Research fields were converted from text labels into numeric values, making them usable for machine learning models.

3.2.1 Handling Missing Values. Before starting the preprocessing, we checked the dataset for any missing values in both the training and test sets. This step is important to ensure that the models are trained on complete data without gaps or empty fields. The check showed that there were no missing values in any of the columns:

```
Train missing values:
 Unnamed: 0    0
 abstract      0
 label         0
 dtype: int64

Test missing values:
 Unnamed: 0    0
 abstract      0
 label         0
 dtype: int64
```

Since no missing data was found, no additional handling was necessary.

3.2.2 Text Cleaning. Before tokenizing the abstracts, we cleaned the text to make it more consistent. First, we removed any entries where the abstract or label was missing. Then, we normalized the text by removing extra spaces, newlines, and tabs. We also converted all text to lowercase to ensure that words like "Physics" and "physics" are treated the same. These steps help simplify the data and improve the performance of later processing and model training.

3.2.3 Tokenization. After cleaning the text, the next step was to split each abstract into individual words, also known as tokens. This process is called tokenization. It helps prepare the text so that models can process each word separately.
Each abstract was turned into a list of word tokens. For example:

Example:

```
clean_abstract: we propose a protocol to encode...
tokens: [we, propose, a, protocol, to, encode]
```

3.2.4 Label Encoding. Since the research field labels are in text form (e.g., "physics", "math"), we needed to convert them into numbers so that they can be used by machine learning models. This process is called **label encoding**. Each unique label is assigned a different integer value.
The mapping between research fields and their corresponding numeric labels is shown below:

Label	Encoded Value
astro-ph	0
cond-mat	1
cs	2
eess	3
hep-ph	4
hep-th	5
math	6
physics	7
quant-ph	8
stat	9

3.3 Train-Dev Split

After preprocessing, we split the dataset into training and development (validation) sets. We used 80% of the data for training and 20% for validation. A stratified split was used, meaning the proportion of each class is maintained in both sets. This helps ensure fair training and evaluation across all categories.

3.4 Summary

In summary, we explored the dataset, ensured the data was clean and well-structured, tokenized the abstracts, encoded the labels, and split the data into training and development sets. These steps prepared the dataset in a reliable and organized way for building and evaluating machine learning models.

4 METHODOLOGY

4.1 Bag-of-Words (BoW) with Feedforward Neural Networks

4.1.1 Feature extraction with CountVectorization: CounterVectorization is a feature extraction method that converts text documents into a matrix of token counts. We are using this method to convert the articles into token vectors. It converts the data into a shape that is suitable to use as input for feedforward Neural Network. Two parameters were passed to the CountVectorizer function as input:

- **max_features=10000:** This limits the words to top 10,000 based on their frequency, reducing the dimensionality of the data and making it less expensive to train. This is also a parameter that can be tuned to extract efficiency during the training process.
- **stop_words='english':** This removes common English words such as 'the', 'and', 'to', etc., which do not contribute to the meaning of the text.

Both training and test sets were applied to the same tokenization process to keep both sets in the same format for testing. The shapes of both sets after the process were as follows.

```
CountVectorizer shape (train): (64000, 10000)
CountVectorizer shape (dev): (16000, 10000)
```

4.1.2 Feature extraction with TF-IDF Vectorization: This is a different tokenization method that converts text documents into a numerical matrix based on their importance. The importance is determined by the frequency of words. **said do you want me to add how it is calculated or not?** The same max_words and stop_words parameters were used just to see the difference between the results of TF-IDF and CountVectorizor. This was also applied to both test and train sets and the shapes of the data were also the same at the end of the process.

4.1.3 Building MLP (For CountVectorization): After the preprocessing and the tokenization phase we moved on to build the feedforward neural network (FeedforwardNN). First we passed the CountVectorizor data to the tensors. The model we created was quite basic just to get a starting point, from there we moved on to fine tune the model by experimenting with the parameters. We experiemented with A different number of hidden layers with different dimensionality, different numbers of epochs, dropout layers, different activation functions. The final architecture of that model is given below.

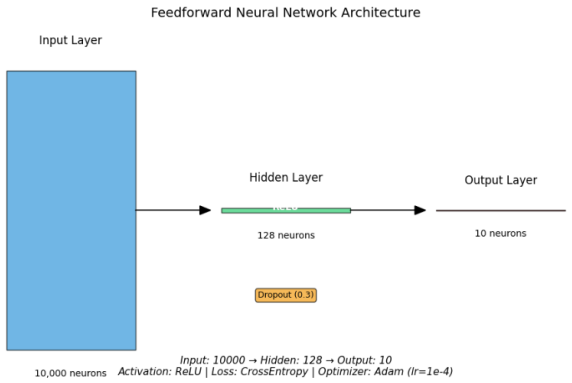

Figure 3: Final shape of the neural network for Countvectorizer fed neural network

4.1.4 Building MLP (For TF-IDF):. We changed the countvectorizor tensors with TF-IDF with the same neural network architect but we saw decline in performance. We decided to tune a different architect for the TF-IDF vectorization. Method was the same to try different parameters to see what impact does it have on the results. Final architecture after tuning is shown in the figure below.

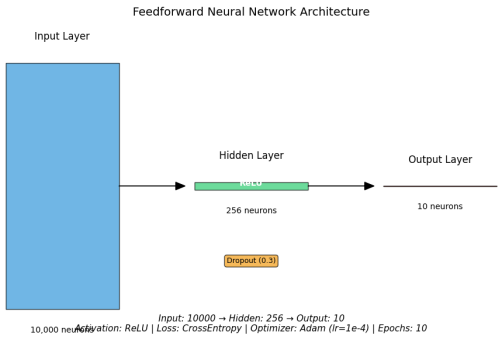

Figure 4: Final shape of the neural network for TF-IDF fed neural network

4.2 Pre-trained Word Embeddings

Now we move on to a completely different approach. Instead of using bag of words, we now create a fixed-sized vector for each abstract by aggregating its word embeddings. We will be using pretrained embeddings called GloVe[2].

To use Glove we first had to download it, It comes in different dimension packages our plan was to use 300d to get the best possible results for the given data.

On the next step we created aggregated word embeddings from the tokens in three pooling. Max, Mean and Sum pooling the plan was to compare the results and find the most efficient and high performing pooling. On training single model on three pooling the results were quite different, we understood to see which pooling gives the best results we would have to tune the models seperatly.

We ended up creating three models for each pooling. Tuning them and comparing the results, complexity and efficiency. First model used mean pooling. We started with a simple model to get a base line and then moved from there to fine tune it. After a long process of experimentation and trying on different number/types of layers, activation functions, dropout layers, batch normalization layers, epochs, learning rates the final model architecture was as its shown in the figure below.

Layer	Output Shape	Act/Drop
Input Layer	(300,)	-
Dense Layer 1	(265,)	ReLU
Dropout 1	-	0.3
Dense Layer 2	(128,)	ReLU
Dropout 2	-	0.3
Dense Layer 3	(64,)	ReLU
Dense Output Layer	(num_classes,)	Softmax

Table 1: Feedforward NN Architecture with mean pooling

Same process was repeated for the second model which used Max pooling. We took the last tuned model as a base and tested max pooling but the results were way off. So we started experimenting and after lots of fine tuning we finalized the model on the following architecture.

Layer	Output Shape	Act / Drop
Input Layer	(300,)	-
Dense Layer 1	(1024,)	LeakyReLU (alpha=0.1)
BatchNormalization 1	-	-
Dense Layer 2	(512,)	ReLU
Dropout 1	-	Rate: 0.5
Dense Layer 3	(265,)	ReLU
Dropout 2	-	Rate: 0.4
Dense Layer 4	(128,)	ReLU
Dropout 3	-	Rate: 0.3
Dense Layer 5	(64,)	ReLU
Dense Output Layer	(num_classes,)	Softmax

Table 2: Feedforward NN Architecture with max pooling

Third model used sum pooling and the final architecture is given in the table below.

Layer	Output Shape	Act / Drop
Input Layer	(300,)	-
BatchNormalization	(300,)	-
Dense Layer 1	(192,)	ReLU
Dropout 1	-	Rate: 0.15
Dense Layer 2	(96,)	ReLU
Dropout 2	-	Rate: 0.15
Dense Layer 3	(48,)	ReLU
Dense Output Layer	(num_classes,)	Softmax

Table 3: Feedforward NN Architecture for Sum pooling

The model using mean was the best performing model out of the three. We will go in detail about the results in an other section below.

4.3 Recurrent Neural Networks (RNNs)

To better understand the structure and meaning of scientific abstracts, we used Recurrent Neural Networks (RNNs). Unlike simpler models like Bag-of-Words, RNNs process the text as a sequence, allowing them to learn from the order of words and capture the context more effectively.

We experimented with four types of RNN-based architectures:

- **Simple RNN**: A basic version that passes information between hidden states, but may struggle with longer sequences.
- **LSTM (Long Short-Term Memory)**: A more advanced model that can retain information over longer distances, helping to avoid the vanishing gradient problem.
- **GRU (Gated Recurrent Unit)**: A simpler alternative to LSTM that trains faster and uses fewer parameters, while still achieving good performance.
- **Bidirectional RNN (BiRNN)**: Wraps a simple RNN layer in both forward and backward directions, providing a more complete context representation by processing the sequence both ways.

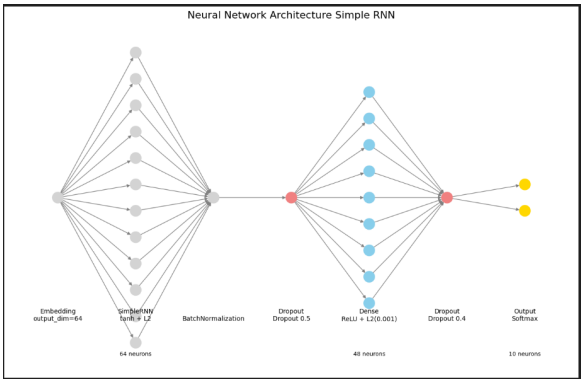

Figure 5: Neural Network Architecture: Simple RNN

The Simple RNN model begins with an embedding layer (output dimension 64), followed by a simple recurrent layer. Batch normalization improves training stability. Dropout layers (0.5 and 0.4 rates) prevent overfitting. A dense layer with 48 neurons captures higher-level features, and the final softmax layer with 10 neurons outputs the predicted class.

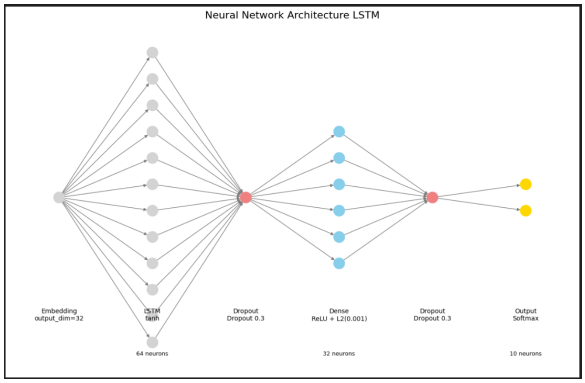

Figure 6: Neural Network Architecture: LSTM

The LSTM model uses an embedding layer (output dimension 32) and an LSTM layer with 64 neurons, allowing the model to remember information across longer sequences. After dropout, a dense layer with 32 neurons refines the features before passing them to the softmax output layer with 10 neurons.

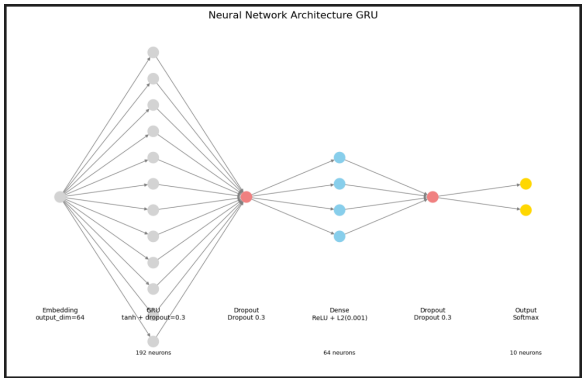

Figure 7: Neural Network Architecture: GRU

The GRU model follows a similar structure. It starts with an embedding layer (output dimension 64), then a GRU layer with 192 neurons, providing strong memory capacity while being faster to train than LSTM. Dropout layers reduce overfitting, and a dense layer with 64 neurons prepares the features for the final 10-class output layer.

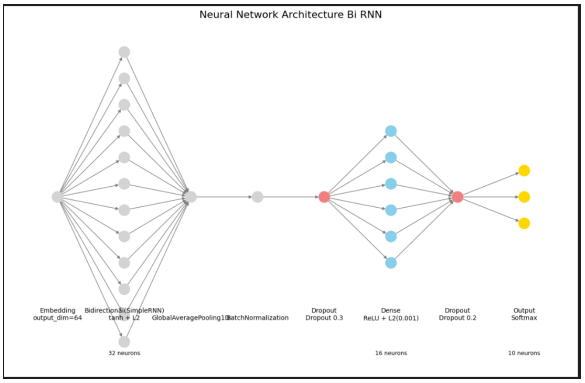

Figure 8: Neural Network Architecture: Bidirectional RNN

The Bidirectional RNN model processes the sequence both forward and backward using a bidirectional RNN (32 neurons in each direction). Global average pooling compresses the sequence output into a fixed-size vector. After batch normalization and dropout, a dense layer with 16 neurons extracts the final features, followed by a softmax layer with 10 neurons for classification.

5 EXPERIMENTS AND RESULTS

5.1 Evaluation of Base MLP models

5.1.1 MLP using CountVectorization: The base model performed at 78% accuracy. From there the **tuning process** was initiated, After running with different parameters and checking the results to get the best possible results the model achieved 84.26% accuracy on validation set in 5 epochs. On the test set the model achieved 83% accuracy which shows that the model generalizing well.

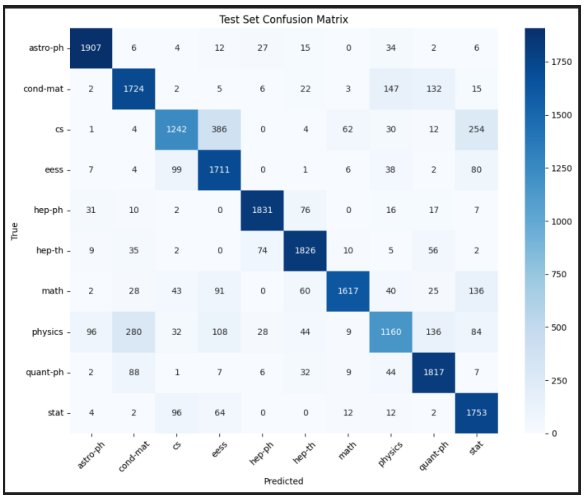

Figure 9: Confusion Matrix of Countvectorizer fed neural network

Reading the confusion matrix it shows on the diagonal which predictions were made accurately. You can also see the classes where the model struggles like class 1 and 2.

5.1.2 MLP using TF-IDF:. After several rounds of tuning we got the results up 84.46% accuracy on validation set which was a tiny bit higher than last model but the model did take more processing time to train. Almost double the last one. On test set the performance was almost the same. Following is the confusion matrix on test set.

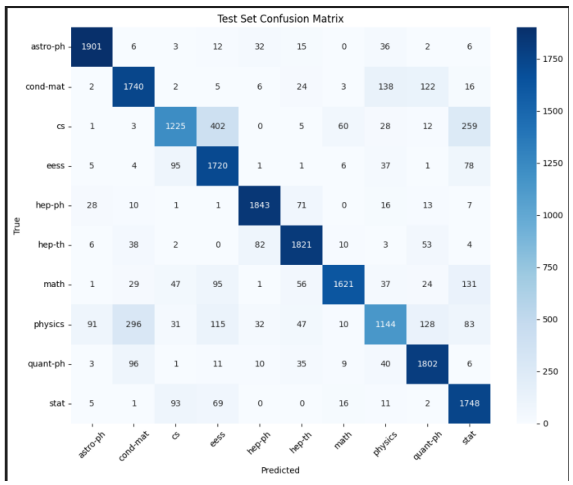

Figure 10: Confusion Matrix of TF-IDF fed neural network

5.1.3 Countvectorizor vs TF-IDF:. Final thoughts on comparing the two we think the simpler approach of countvectorizer works the best in this scenario.

5.2 Evaluation of Models using pretrained embeddings

After lots of fine tuning the training accuracy reached 78.5% and validation accuracy reached 78.1% accuracy. Heres a diagram of model going through epochs.

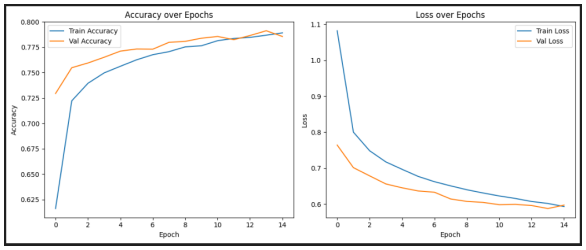

Figure 11: Mean pooling through epochs

Second model was tuned using Max pooling. Final result after tuning was at 72.9% test accuracy and validation accuracy 73.1%, the max pooling data was constantly underfitting. It required a very complex model and more computation time to achieve worse results. Here's a diagram of how this model learns through the epochs.

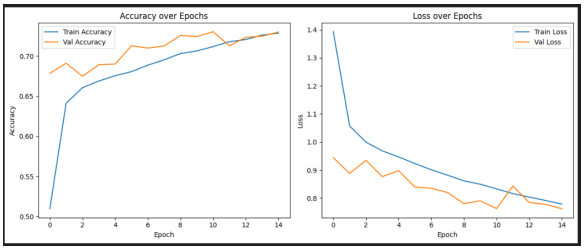

Figure 12: Max pooling through epochs

Last model was tuned using Sum pooling. It started as an overfitting model on the baseline but after lots of trial and error. It achieved 79.5% accuracy on validation set and 79.9% test accuracy. The model was tuned to generalize well. The training diagram is given below.

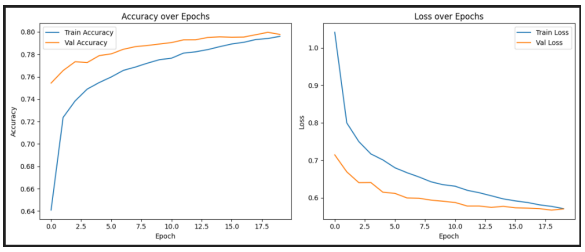

Figure 13: Sum pooling through epochs

Final thoughts about using pretrained embeddings, We achieved the best result using simple BOW process which was cheap computationally. But if we compare Max, Mean and sum pooling models. The mean pooling data performs the best with the simplest model and the best learning curve.

5.3 Evaluation of RNN Models

We trained and evaluated four RNN variants using their respective training and validation accuracy and loss curves, and compared their final test accuracies.

5.3.1 Simple RNN. The Simple RNN exhibited steady improvement during training and achieved a test accuracy of **72.68%**.

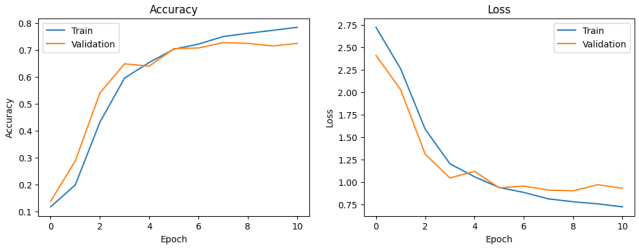

Figure 14: Simple RNN — Accuracy and Loss over Epochs

5.3.2 LSTM. The LSTM model performed well, reaching a test accuracy of **78.72%**. Its training and validation accuracy rose consistently while losses decreased smoothly.

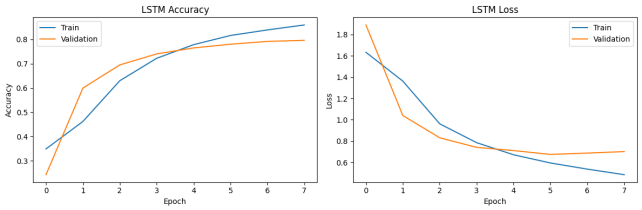

Figure 15: LSTM — Accuracy and Loss over Epochs

5.3.3 GRU. The GRU converged rapidly and maintained stable performance, achieving a test accuracy of **79.61%**.

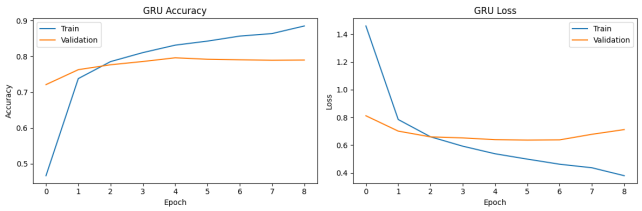

Figure 16: GRU — Accuracy and Loss over Epochs

5.3.4 Bidirectional RNN. The Bidirectional RNN also performed strongly with a test accuracy of **78.53%**. It showed high training accuracy but a slightly earlier plateau on the validation set.

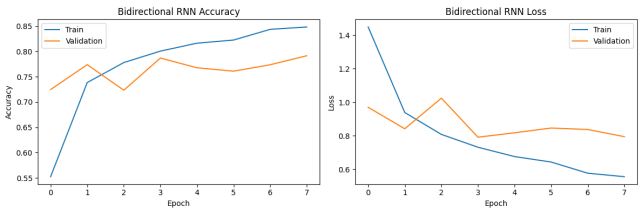

Figure 17: Bidirectional RNN — Accuracy and Loss over Epochs

5.4 Model Comparison

Figure 18 presents the final test accuracies of all four models. The GRU achieved the highest accuracy, followed closely by LSTM, then bidirectional RNN, and lastly the Simple RNN.

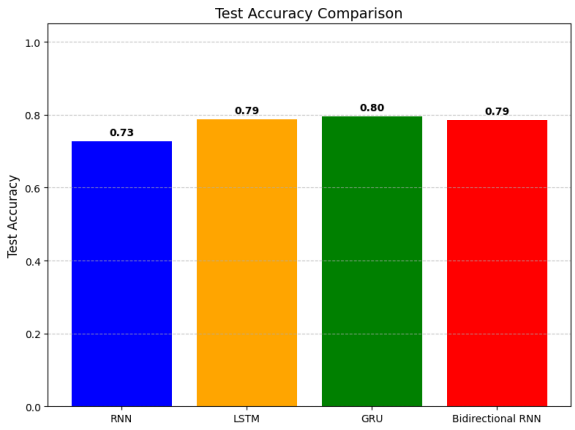

Figure 18: Test Accuracy Comparison Across RNN Models

Overall, while all architectures generalized well, the GRU showed the best test performance for this task.

6 DISCUSSION

In this work, we evaluated three modeling strategies for sorting scientific abstracts into ten categories: a simple feedforward network on Bag-of-Words, a similar network using averaged GloVe embeddings, and several recurrent neural networks (Simple RNN, LSTM, GRU, and bidirectional RNN). The Bag-of-Words approach proved both efficient and accurate, achieving about 83% test accuracy with very little training time.

By contrast, the GloVe-based models reached around 79% accuracy and required longer training, indicating that for short texts (100–200 words), counting word occurrences often captures the most critical signals. Within the RNN group, the GRU slightly outperformed the LSTM (approximately 79.6% vs. 78.7%), while bidirectional and plain RNNs offered only marginal gains despite their added complexity.

Error analysis showed that related fields like computer science and condensed matter physics were frequently confused, reflecting shared terminology in their abstracts. Very brief summaries (under 100 words) also led to more mistakes, suggesting that limited context hinders accurate classification.

These findings underscore that when compute resources or labeled data are scarce, a straightforward Bag-of-Words model can serve as an excellent baseline.

7 CONCLUSION AND FUTURE WORK

We compared several neural network methods for classifying scientific abstracts into ten fields. The simplest approach—counting words and using a small feedforward network—reached about 83% accuracy and trained very quickly. Switching to averaged GloVe embeddings gave around 79% accuracy but required more training time.

Among recurrent models, the GRU performed best (nearly 80%), while LSTM and bidirectional RNNs only added small improvements. We also saw that related fields, such as computer science

and condensed matter physics, were often confused, especially when abstracts were shorter than 100 words.

In future work, we will fine-tune transformer models like BERT or SciBERT to capture context better. We plan to train embeddings on a large collection of scientific papers so the model learns technical terminology. Adding extra information, such as paper titles and author keywords, could help the model make clearer distinctions.

Finally, we will explore combining different model types and applying stronger regularization to avoid overfitting. These steps should lead to a faster, more accurate classifier for scientific abstracts.

REFERENCES

[1] J. Glenn and M. Goldman. 1976. Arxiv-10 Dataset. Unpublished manuscript.
[2] Christopher D. Manning Jeffrey Pennington, Richard Socher. 2014. GloVe: Global Vectors for Word Representation. https://nlp.stanford.edu/projects/glove/