Autokeras Text Classification - Final Report

Cesar Lopez cesarhn95 224005226

Praveen Venugopal razorvine 727006043

Vansh Narula narulavansh3 128001615

Abstract

Binary classifiers are used to predict the sentiment behind a give sentence (meaningful sequence of words). One of these famous examples is the IMDb (1) model in which a probabilistic approach is used for determining if a movie review is overall positive or negative. A more recent approach to doing sentiment prediction is by using Neural Networks, specially RNNs. This approach is commonly called Natural Language Processing or NLP. By using a bidirectional LSTM we were able to properly obtain similar sentiment prediction accuracy than previous purely statistic models. However, LSTMs are not trivial, they require the raw input to be embedded, and the size and shape of the model will yield different results and accuracies for a given input. Searching through the best embedding sizes and best model hyperparameters for a model given an input is not a trivial task. Neural Automated search within the Auto-keras framework was used such that hyperparameter search for a given input is trivialized. The end result was an auto-keras models that will look through the best hyperparameters for this LSTM model given an input dataset.

1 Introduction

1.1 Sentiment Analysis

Sentiment Analyisis is the area of linguisitcs that categorizes a statement having a positive or negative connotation. A simple binary classifier, although seen simple and commonly called the Hello World of NLP, sentiment prediction has a wide range of uses ranging from email spam filter to performing automated review de-noising. Using examples from the IMDB dataset (1) This problem might seem trivial from clear examples like:

- [**POS**] This movie 's AWESOME
- [**POS**] TRUE ART !!!
- [**NEG**] Detestable Kubrick social commentary I hold in high disdain .
- [**NEG**] Stupid , stupid , stupid

This example can be properly categorized by recognizing keywords which have posititve or negative connotation. For example words like "Art" and "Awesome" are positive, while "detestable" "stupid" are negative. This model however is easy to fool and incredbibly unreliable. This was the premise for Weizenbaum's ELIZA which would suggest the context and the word topics based on keywords. One can see how quickly this example can be turned on its head by the following additions:

- [**NEG**] This movie 's not AWESOME *(double negative)*
- [**NEG**] Wannabe Art, pretentious (sarcasm, ironic tone)
- [**POS**] This situation is Detestable, how did Kubrick did not do social commentary before? *(negative terms are abundant)*

- [**POS**] "Momma says stupid is what stupid does" hahaha *(New term that only belongs to this movie)*

We can see that by quick additions of words, one can flip the polarity of any given statement. As such sentiment analysis is impossible to handcraft. Enter Machine learning. Probabilistic models range from Bag of Words (introduced in the 50s by Zellig Harris) to more complex machine learning protocols that parse through an entire document and quantize parameters between words like the one introduced by Maas et. all (1). For example, the model from Maas et all. scans the documents and extracts *semantic similatities* then word sentiment then trains a model that maximizes prediction accuracy. In order to do proper testing Maas et all (1) created the IMDb dataset (which we used for our initial testing).

After the famous Khrizhevzy result (2) a revival and the availability of more data, Neural Network based mechanisms (in particular those relying on Neural Networks) have been applied to natural language problems like sentiment analysis. Although initially designed for image processing and categorization, Neural Networks like recursive neural networks based on long short-term memory blocks (typically called LSTM) have gained popularity within text classification solutions. With their benefits, RNN models come with their drawbacks, depending on the wanted accuracy, some models with different shapes and sizes will not be able to achieve a a wanted accuracy.

Recurrent Neural networks as seen on figure **??** are a special type of convolutional neural network that will feed the output of a layer into the input of the next data point. This is helpful because it grants the network the ability of processing data on a element by element basis with the ability of keeping track of the elements that came before it. As a result a RNN has the ability of granting the network non-volatile memory. This is specially helpful for Language models because we can process each sentence on a word by word basis and our network would keep track of the words that came before it.

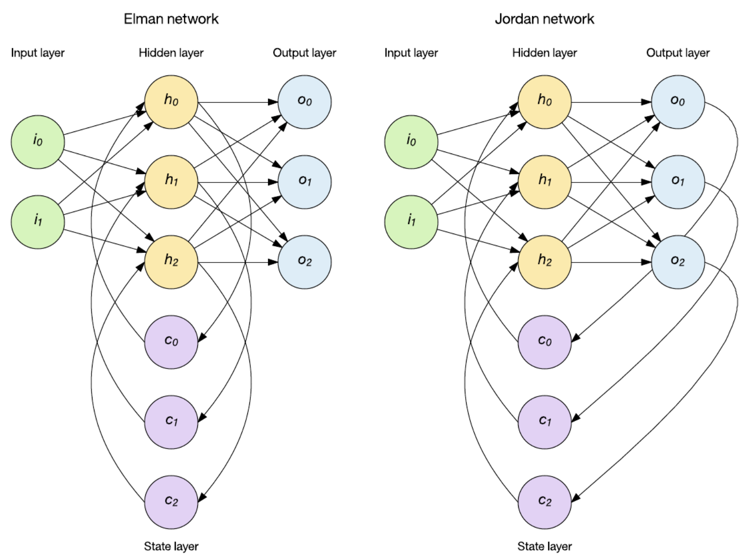


Figure 1: Examples of Recurrent Neural Networks
c

As with other types of neural networks, RNN expect an input on a specific format with size. Given the variation in the length and possible characters in words, it is not possible to feed raw words into an RNN. Word Embeddings such as Words2Vec were initially introduced by Milokov et al. (3) as a way to transform words into a uniform vector of their features, so they can fed into an RNN. As such, word embeddings are a make it or break it part of neural networks.

1.2 Automated Machine Learning

With the new-found popularity of machine learning mechanisms, applications are growing faster than data scientists available for executing them. Automated machine learning (or AutoML) aims at abstracting the intricacies of different machine learning models from the final user. But, not all models will be optimal for all types of data. AutoML approaches this by doing neural automated search. The end goal of NAS is to perform a search inside a range of given tunable hyperparameters to get the model that is optimal for a given data.

Given the popularity of the Keras framework in CNN-based machine learning models, Jin et all (4) created Auto-Keras as an AutoML framework based on Keras. Auto-keras aims at further democratizing the power of Machine Learning by being a completely open source framework. This allows the user to create an refine a model from the privacy of their local system. Although other popular closed source AutoML projects like Google's Cloud AutoML have been very popular, they are not compatible with applications in where sensitive data cannot be sent and processed in the cloud. In addition, having the comfort to create an optimal model for your data from your local system allows the model to run and retrain without the availability of an internet connection. The applications are endless.

In this work we will aim at creating a generic Binary Sentiment classifier that can identify the correct sentiment of a statement and include it as a supervised class for the Auto-Keras framework. In this way, once included in a framework, our model will search for the best hyperparmeters given a generic input and generate the best model for it.

2 Related Work

2.1 Early Approaches

The nuances associated with the term Machine Learning, paints an impression that these problems are quite recent. But the fact would be that such challenges existed for decades and novel approaches have been suggested to tackle them. One of teh earliest work on text classification applied a Naive Bayes approach for classification of sentences. Andrew McCallum et all. (5) use a multinomial model to capture word frequencies and calculate probability of occurrence of subsequent words. The paper used Precision/Recall as performance measure. Further advances were done by Kamal et al. in 1999 (6) using Expectation Maximization , and by Simon Tong et al. in 2001 (7) using Support Vector machines. These papers give us a rich insight to teh prior work done with respect to Text classification under more pristine settings when Neural Networks were still not in teh big screen.

The original plan for doing text classification submitted on previous project proposal consisted of applying a *Universal Language Modeling* for text classification. This approach consists on pre-training a model with a large amount of generic data. In this way an initial pre-trained model will contain information about what connotation each word evokes (can vary from binary class to k classes). Once the model is trained for every possible word, the model is fine tuned with data specific to the task we want our model to be tested on.

After fine tuning, training is redone by doing *gradual unfreezing*. On this technique our new model is retrained one layer at a time in order to train our model from more general to least general. This is the only approach that the authors mention as adaptable for other models.

Although proven to be effective, this model requires access to the original words and cross referencing them with the model that we are going to train (IMDb dataset in our case). This does not align with the mode of operation for auto-keras (4) in where the model has to have little or no information about the data it processes. The ideal auto-keras text classifier will encode the words into different integer IDs in each sentence and our model should be able to draw data from it. As such our Technical plan had to be adapted into these requirements.

2.2 Deep Learning approaches

Roughly a decade later, Xavier Glorot et al. (**?**) submitted their work wherein they implemented a classifier using denoising Autoencoders Post this there were many experimentation done with different variant of Neural networks which included CNN, RNN and RCNN. Recurrent layers showed good results in theis field of text classification as proved by Pengfie et al (8) and Chunting et al (9). This lead to a phase were many papers experimented with variants of RNN such as GRUs and LSTM . THese paapers although showed good accuracy and results, did not introduce any staggering changes to teh overall approach. CNNs made a comeback in 2016 in the work of Alexis Conneau et al (10), but soon were overshadowed by recurrent networks.

2.3 Universal Language Modelling - ULM

While Deep Learning models have achieved state-of-the-art on many NLP tasks, these models are trained from scratch, requiring large datasets, and days to converge. Research in NLP focused mostly on transductive transfer (Blitzer et al., 2007)[https://www.cs.jhu.edu/ mdredze/publications/sentiment_acl07.pdf]. For inductive transfer, fine-tuning pre-trained word embeddings (Mikolov et al., 2013), a simple transfer technique that only targets a model's first layer, has had a large impact in practice and is used in most state-of-the-art models. Recent approaches that concatenate embeddings derived from other tasks with the input at layers . This proves to the effectiveness of using pre-trained models. In our work, we combine the proven abilities of LSTM models with pre trained word embeddings to achieve a good accuracy on teh given dataset.

3 Methodology

3.1 FC layers and CNN

Our first approach to sentiment prediction was performing word embeddings with our corpus and creating the neural network that preceeds it from scratch. The initial plan consisted on implementing a Universal Language Model using LSTM. Although this model is novel and achieves great performance, its implementation was non-trivial. The first phase pertaining to general domain pre-training of the Language Model requires huge train data set, at least comparable in size with Wikitext-103. Although initially a type of word embedding using TF-IDF/One hot encoding was tried, another large dataset had to be used for initial embedding and it had to share encoding with the targeted data (IMDb in our case) and this was something non feasible with the given data under the time. Given that the course project needed to pass the benchmark set by auto-keras sample train and test data sets which are rather smaller in size, pre-training on those data large sets was deemed non feasible.

Parting from UMLFit, we tried to regularize each sentence by turning each word into a one-hot encoded vector and with a vector the size of the maximum amount of words, we fed it into a vanilla fully connected layer, as this was the topic we were learning in class at the moment. Initial testing, even after hyperparameter search showed that the accuracy was not getting higher than 50%, which mean that we might as well be giving random noise to our neural network because it couldn't learn. We had not yet realized that this approach does not translate well because there is no amount of recurrence and the result of processing a word will have no effect on the word processed after it.

The next stage we tried was using Convolutional neural networks. We did one hot encoding for the entire sentence. We started this time by using the raw dataset and manually building a vocabulary that mapped an ID to each new word. Then we one-hot encoded a sentence to a 2D array of size m x n where m is the maximum number in the vocabulary and n is the maximum length of a sentences, each n position correspoding to a word will have 1 for the position of the ID its word belonged. Then we connected this to a vanilla CNN. This approach was tried when we were learning CNNs in our class. Although this seemed promising initially, because we kept the encoded words in their order and the words were mapped one to one with our created dictionary, this also did not work, again achieving an accuracy of 50%. The One-hot matrix created was incredibly sparse and any possible data embedded in the form of an ID matching to a word was quickly diluted vanished after doing convolution with neighbors full of zeroes.

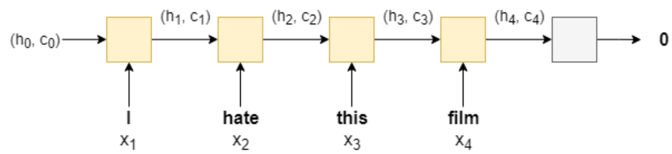
After our first two tries and knowing that we were not making any progress, and training could not be done in a generic encoding or manually encoding setting, it was evident that the first step of our algorithm had to be finding a usable encoding scheme. This also coincided with the time in class that we started learning about Recurrent Neural Network.

3.2 Multi layer Bi-LSTM Model

Post milestone One, our experimentation with Recurrent Neural Network began picking up pace. As we saw earlier that RNN has a structure to capture time-series data, it still is not efficient in retaining the information int the long run. Especially for long sentences this drawback hurt performance drastically. Hence we use a specific variant call Long Short Term memory. Standard RNNs suffer from the vanishing gradient problem. LSTMs overcome this by having an extra recurrent state called a cell, c - which can be thought of as the "memory" of the LSTM - and the use use multiple gates which control the flow of information into and out of the memory. For more information, go here. We can simply think of the LSTM as a function of x_t, h_t and c_t, instead of just x_t and h_t.

$$(h_t, c_t) = LSTM(x_t, h_t, c_t)$$

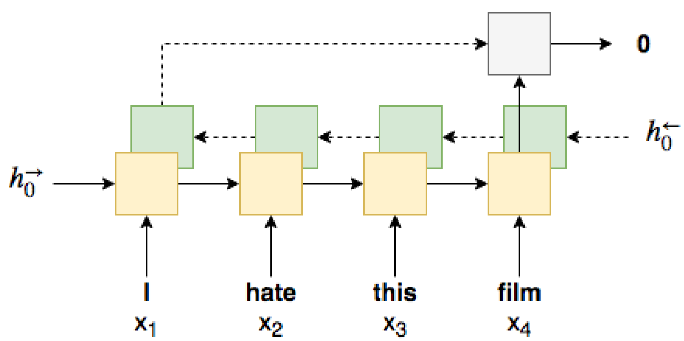
Thus, the model using an LSTM looks something like:



The initial cell state, c_0, like the initial hidden state is initialized to a tensor of all zeros. The sentiment prediction is still, however, only made using the final hidden state, not the final cell state, i.e. $\hat{y} = f(h_T)$

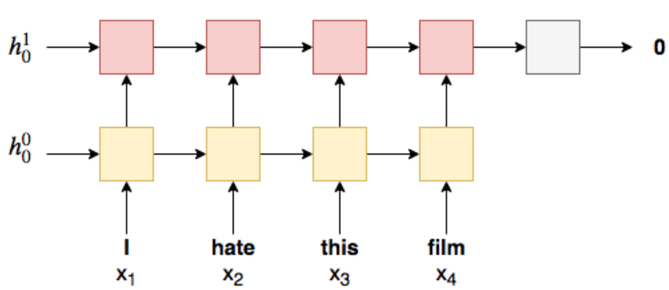
Next we tried with Bi-directional RNN. Herein, in addition to having an RNN processing the words in the sentence from the first to the last (a forward RNN), we have a second RNN processing the words in the sentence from the last to the first (a backward RNN). In PyTorch, the hidden state (and cell state) tensors returned by the forward and backward RNNs are stacked on top of each other. We make our sentiment prediction using the last hidden state from the forward RNN (obtained from final word of the sentence), $h_T^{\rightarrow}$, and the last hidden state from the backward RNN (obtained from the first word of the sentence), $h_T^{\leftarrow}$, i.e. $\hat{y} = f(h_T^{\rightarrow}, h_T^{\leftarrow})$

The image below shows a bi-directional RNN, with the forward RNN in orange, the backward RNN in green and the linear layer in silver.



Multi-layer RNNs (also called deep RNNs) are another simple concept. The idea is that we add additional RNNs on top of the initial standard RNN, where each RNN added is another layer. The hidden state output by the first (bottom) RNN at time-step t will be the input to the RNN above it at time step t. The prediction is then made from the final hidden state of the final (highest) layer.

The image below shows a multi-layer unidirectional RNN, where the layer number is given as a superscript. Also note that each layer needs their own initial hidden state, h_0^L.



.

3.3 Pre-trained Word embeddings

After successful understanding and implementation of the use cases of LSTM, or RNN in general, for time series data, we realized that although this helped increasing the accuracy a bit but changing the architecture alone would not yield the best results. The network alone was not able to learn the contextual semantics of the sentences which is the key for a good sentiment classification algorithm. Hence, post this we dedicated our time to fetch good vector representation of our initial raw data. We came up with an option to use pre-trained word embeddings. Now, instead of having our word embeddings initialized randomly, they are initialized with some pre-trained vectors. The theory is that these pre-trained vectors already have words with similar semantic meaning close together in vector space, e.g. "terrible", "awful", "dreadful" are nearby. This gives our embedding layer a good initialization as it does not have to learn these relations from scratch.

While researching about the same, we came across GloVe : Global Vectors for Word Representations, which is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.

The GloVe model is trained on the non-zero entries of a global word-word co-occurrence matrix, which tabulates how frequently words co-occur with one another in a given corpus. Populating this matrix requires a single pass through the entire corpus to collect the statistics. For large corpora, this pass can be computationally expensive, but it is a one-time up-front cost. Subsequent training iterations are much faster because the number of non-zero matrix entries is typically much smaller than the total number of words in the corpus.

We used "glove.6B.100d" vectors". 6B indicates these vectors were trained on 6 billion tokens and 100d indicates these vectors are 100-dimensional. Now, After obtaining our word representations, we pass them as inputs to our Bi-Directional LSTM network.This resulted in a tremendous boost in our model prediction accuracy. The accuracy boost was solely because the pretrained vectors helped the model to learn the contextual relationships much better and thus, Now, the model can classify the sentence using the hidden sentimental relationship between the words in sentences rather than the words itself.

3.4 Hyperparameters

Once our LSTM model was properly connected to the datasets given, we proceeded to doing NAS. For hyper-parameters, we saw initially that changing the maximum vocabulary will reduce the score at lower numbers while increasing compute time for higher ones. Hence we selected the maximum vocabulary size as our first tunable hyperparameter. Number of Epochs, hidden dimensions, dropout and number of layers, are the most common hyperparameters changed in order to increase performance. Performance was a function of accuracy. Independent of the search algorithm used, the score, along as a set of hyperparameters used to obtain it was stored. When hyperparameter search was exhausted or the time limit was surpassed, the fit function selects and prints the best hyperparameters and stores them internally inside the text classifier class. The range for each hyperparameter are given in table 1. The colloquially coined "Grad Student Search" was used initially to find these parameter (which consists on trying different options and then manuallly picking the best ones.

hyperparameter	range
hidden dimensions	64, 128, 256
batch size	16, 32, 64
number of epochs	2, 3, 5
maximum vocabulary size	20000, 25000
number of hidden layers	2, 3, 4
dropout	0.4, 0.5, 0.6

Table 1: Range for final Greedy search

3.5 Grid Search

The simplest way of performing hyperparameter optimization has been grid search, or a parameter sweep, which is simply an exhaustive searching through a manually specified subset of the hyperparameter space of a learning algorithm. A grid search algorithm must be guided by some performance metric, in our case prediction accuracy in a held-out validation set. Since the parameter space of a machine learner may include real-valued or unbounded value spaces for certain parameters, manually set bounds and discretization may be necessary before applying grid search.

Grid search suffers from the curse of dimensionality, but is often embarrassingly parallel because typically the hyperparameter settings it evaluates are independent of each other.

3.6 Random Search

This is another simple way of choosing hyperparameters where we randomly generate indexes for all the possible hyperparameters one after another. Then we best the model with the obtained set of hyperparameters and does the same after the training stops for the previous model. We keep a track of the model parameters and the corresponding validation accuracy and thus make sure that the same parameters are not repeated again and again. Thus, training with random combination of hyperparameters at each training cycle until all the possible combinations have been tested or the time limit is exceeded. In this we hope that there is chance that our best model will get trained before the time limit is exceeded and thus, this relies on the probability of selecting the best model out of many possible models. This probability becomes very small as the search space becomes larger. But we can still rely on the fact that the model will somehow reach near to best model parameters in the given time limit. Thus, sometimes random search gives better result than grid search when either search space is very large or the dimensions of the grid is huge because there are many tunable parameters.

3.7 Naive Greedy Search

To complete the final step of the project, we needed to come up with a smarter search algorithm than grid or random search. So, we came up with a Naive Greedy Search which is a combination of Grad student search and Greedy search where we do not provide the entire space to the greedy search. Rather, we stick to a smartly defined search space based on our understanding of the network and then we proceed with greedy search in that space. Hence, doing this, we are able to reduce the search space significantly. This allows us to traverse through the solutions that are most relevant without going over all the possible hyperparameters like in grid search. For each hyperparameter we traverse through the smartly defined range and select the hyperparameter which gives the best validation accuracy. Once we have the hyperparamater in hand, we fix its value and move onto the next hyperparameter. This way, we perform a naive greedy search for each tunable hyperparameter where the sequence of the hyperparameters is also defined usig the Grad student approach where we mimic human thinking process while dealing with hyperparameters and its sequence of selection using greedy search.

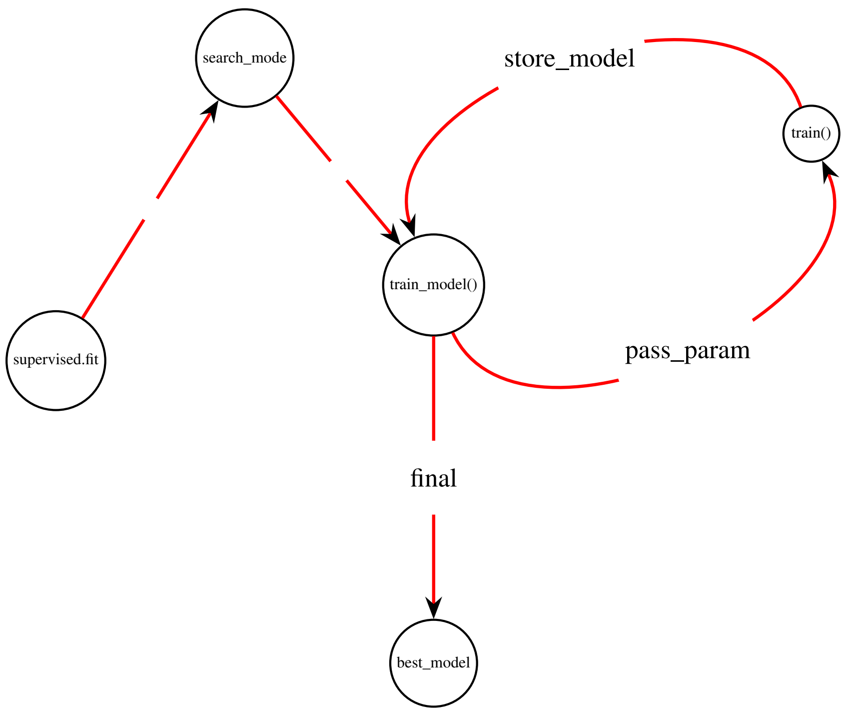
4 Results

4.1 Code Walkthrough

Our current implementation uses torchtext framework to load dataset and iterate over it. The code dynamically generates tsv files from the given numpy train data. It is this tsv file that is loaded as torchtext dataset, and iterated over. Two global torchtext data fields are created one to hold the input sentence and the other to hold output sentiment of the train data. We build vocabulary for the data fields only using the train data, during model training.

Word-Embeddings are another crucial aspect of this section. We use Glove.6B package with a size of 100-dimensions. This converts the individual word in the sentence to a vector of 100-D. Consequently our embedding_layer has a size of 100, throughout the project.

Finally, each search algorithm is either exhaustive (will not repeat a search) and has a soft stop for a time limit passed. This means that once the time runs out, fit will finish running the model that it is training before searching for the best one. Once the search is done, fit will select the best model from the entire search history, even when doing grid search. This ensures that even if a bad decision is made by Greedy search, we can always go back to our best previous state.



4.2 Initial Results with final model

In order to do initial testing, we limited our runs to do the small 96k + 4k raw dataset. After veryfyin that we could train and keep different models, we selected a 332 confgiguration. Given that grid search and random search are lengthy to exhaust, we limited our search space to 3 values of hidden dimensions, 3 values of of batch sizes and 2 values of number of layers. With this configuration, both grid and random will calculate 6*6*3 , models with a total of 18, and greedy will calculate 3+3+2 = 8. We can see that Greedy converges faster, although it is not exhaustive. The progression of these algorithms on a model by model can be seen in tables 5, 5 and 5 (present in the appendix). We can see on table 5 that the random search is exhaustive (no repeated values) and each new set of values is selected in a random order. for grid in table 5 we can see that we explore every possible choice of each value. Finally, for greedy search 5 we can see that we selected the best hyperparameter for hidden dimensions, with this set, we went on and searched for the best batch size and finally the same for number of hidden dimensions.

4.3 Final Results with final model

Once we were comfortable that Greedy search was as efficient as the other two but with an exponentially smaller run time (due to its smart reduced search space) a 3-3-2-3-3-3 search was done. For all parameters except max vocab size, we selected 3 possible values. Having set our final search space, we proceeded to get the best hyperparameters through grid search for both the small raw dataset (96k +4k) and the raw large dataset (480k +). The results of the run for the small dataset are shown in the appendix as table 5. The final results for the large dataset are shown in table 4.3. In here we can see that the greedy in runs 1,2,3 converges that best accuracy is obtained at 3 layers, then in runs 4,5,6 it learns that the best number of hidden dimensions is 256 and the same thing happens for the other hyperparameters.

Run	Accuracy	hidden_dim	batch	epoch	vocab	layers	dropout	time
1	83.65	128	16	2	20000	2	0.4	418.58
2	83.70	128	16	2	20000	3	0.4	593.71
3	83.67	128	16	2	20000	4	0.4	847.53
4	83.43	64	16	2	20000	3	0.4	515.43
5	83.54	128	16	2	20000	3	0.4	594.29
6	83.68	256	16	2	20000	3	0.4	929.74
8	83.63	256	16	2	25000	3	0.4	943.97
9	83.58	256	16	2	20000	3	0.4	926.80
10	83.48	256	32	2	20000	3	0.4	684.94
11	82.81	256	64	2	20000	3	0.4	546.93
12	83.76	256	16	2	20000	3	0.4	930.28
13	84.44	256	16	3	20000	3	0.4	1394.24
14	84.80	256	16	5	20000	3	0.4	2318.21
15	84.88	256	16	5	20000	3	0.4	2314.91
16	84.18	256	16	5	20000	3	0.5	2306.06
17	83.46	256	16	5	20000	3	0.6	2301.99

Table 2: Results for greedy search 5 tunable parameters, 3|3|2|3|3| options for 480k dataset

5 Conclusion

After days of extensive learning and testing we have come up with a final Multi-layer Bi directional LSTM model that performs Naive Greedy Search for hyper parameter optimization for sentiment classification. This uses pretrained GloVe embedding to learn the contextual relationship between the words in a sentence and then classifies the sentence based on this relation. This model makes use Auto-Keras Supervised class and hence can be included inside the original Auto-Keras code with potential improvements in search strategy and word embedding, which at last is the room for future improvement that is always there in any solution. The final accuracy achieved after the greedy search on the final test data set is 85.4% and the chosen hyperparameters are 256 hidden dimensions, 16 batch size, 5 epochs, vocab size 20k, 3 layers and 0.4 dropout. The parameters and accuracy are not fixed and can be improved with larger search space but that will result in increased search time. this, based on our understanding and skill set, is the most efficient way to automate text classification task in the given time frame.

Work Distribution and Acknowledgement

Each and every member performed and contributed equally to all the tasks. The work was divided but was never completed by anyone alone and hence, the team actually showed great amount of collaborative efforts which towards the end equipped each one with a broader understanding of the task in hand. The whole project was looked upon as an opportunity to grow and learn the basics of NLP and the use cases of Neural Networks in solving such tasks. We are glad the outcome was as expected.

We would like to express our gratitude to our mentor Haifeng Jin and professor Shuiwang Ji for always being there and guiding us throughout this phase of learning. We thank you for providing us with the task and considering us capable enough to complete it in the given time frame. We hope we have done justice to your expectations.

References

[1] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, (Portland, Oregon, USA), pp. 142–150, Association for Computational Linguistics, June 2011.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, (USA), pp. 1097–1105, Curran Associates Inc., 2012.

[3] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *CoRR*, vol. abs/1310.4546, 2013.

[4] H. Jin, Q. Song, and X. Hu, "Auto-keras: Efficient neural architecture search with network morphism," 2018.

[5] A. McCallum, K. Nigam, *et al.*, "A comparison of event models for naive bayes text classification," in *AAAI-98 workshop on learning for text categorization*, vol. 752, pp. 41–48, Citeseer, 1998.

[6] K. Nigam, A. K. Mccallum, S. Thrun, and T. Mitchell, "Text classification from labeled and unlabeled documents using em," *Machine Learning*, vol. 39, pp. 103–134, May 2000.

[7] S. Tong and D. Koller, "Support vector machine active learning with applications to text classification," *J. Mach. Learn. Res.*, vol. 2, pp. 45–66, Mar. 2002.

[8] P. Liu, X. Qiu, and X. Huang, "Recurrent neural network for text classification with multi-task learning," *CoRR*, vol. abs/1605.05101, 2016.

[9] C. Zhou, C. Sun, Z. Liu, and F. C. M. Lau, "A C-LSTM neural network for text classification," *CoRR*, vol. abs/1511.08630, 2015.

[10] A. Conneau, H. Schwenk, L. Barrault, and Y. LeCun, "Very deep convolutional networks for natural language processing," *CoRR*, vol. abs/1606.01781, 2016.

Appendix

Accuracy	hidden_dim	batch	epoch	vocab	layers	dropout	time
76.00	64	4	1	25000	2	0.5	140.29
75.95	128	4	1	25000	2	0.5	142.93
76.05	256	4	1	25000	2	0.5	174.57
76.32	256	4	1	25000	2	0.5	171.85
74.94	256	8	1	25000	2	0.5	98.93
75.12	256	12	1	25000	2	0.5	74.59
76.43	256	4	1	25000	2	0.5	174.60
76.10	256	4	1	25000	3	0.5	248.21
76.23	256	4	1	25000	2	0.5	173.46
75.28	256	4	1	25000	3	0.5	246.35
75.89	256	8	1	25000	2	0.5	99.94
75.20	256	8	1	25000	3	0.5	146.14
74.82	256	12	1	25000	2	0.5	74.92
75.13	256	12	1	25000	3	0.5	112.04

Table 3: Results for grid search 3 tunable parameters, 3|3|2 options

Random Search

Accuracy	hidden_dim	batch	epoch	vocab	layers	dropout	time
79.41	128	16	2	20000	2	0.4	86.73
79.08	128	16	2	20000	3	0.4	119.62
79.35	128	16	2	20000	4	0.4	169.30
79.30	64	16	2	20000	2	0.4	78.50
79.77	128	16	2	20000	2	0.4	85.07
79.68	256	16	2	20000	2	0.4	123.00
79.83	128	16	2	20000	2	0.4	85.41
79.08	128	16	2	25000	2	0.4	88.43
79.37	128	16	2	20000	2	0.4	82.77
78.30	128	32	2	20000	2	0.4	54.00
76.92	128	64	2	20000	2	0.4	39.30
79.27	128	16	2	20000	2	0.4	85.55
80.66	128	16	3	20000	2	0.4	126.40
81.76	128	16	5	20000	2	0.4	208.95
81.82	128	16	5	20000	2	0.4	208.94
81.08	128	16	5	20000	2	0.5	208.97
80.21	128	16	5	20000	2	0.6	209.16

Table 4: Results for random search 3 tunable parameters, 3|3|2 options

Greedy Search

Accuracy	hidden_dim	batch	epoch	vocab	layers	dropout	time
76.00	64	4	1	25000	2	0.5	140.29
75.95	128	4	1	25000	2	0.5	142.93
76.05	256	4	1	25000	2	0.5	174.57
76.32	256	4	1	25000	2	0.5	171.85
74.94	256	8	1	25000	2	0.5	98.93
75.12	256	12	1	25000	2	0.5	74.59
76.43	256	4	1	25000	2	0.5	174.60
76.09	256	4	1	25000	3	0.5	248.21

Table 5: Results for greedy search 3 tunable parameters, 3|3|2 options

Accuracy	hidden_dim	batch	epoch	vocab	layers	dropout	time
76.37	128	4	1	25000	2	0.5	143.75
73.74	64	8	1	25000	2	0.5	72.94
75.57	128	12	1	25000	2	0.5	53.48
74.53	64	12	1	25000	3	0.5	66.62
75.59	64	12	1	25000	2	0.5	50.92
74.82	128	12	1	25000	3	0.5	73.26
75.72	64	8	1	25000	3	0.5	93.97
76.07	64	4	1	25000	2	0.5	138.90
75.98	128	4	1	25000	3	0.5	184.14
75.20	256	8	1	25000	3	0.5	146.13
75.38	128	8	1	25000	3	0.5	100.64
76.23	256	4	1	25000	2	0.5	173.49
75.27	256	4	1	25000	3	0.5	246.34
75.88	256	8	1	25000	2	0.5	99.94
74.81	256	12	1	25000	2	0.5	74.92
76.08	64	4	1	25000	3	0.5	175.36
75.40	128	8	1	25000	2	0.5	75.98
75.12	256	12	1	25000	3	0.5	112.03

Table 6: Results for greedy search 6 tunable parameters, 3|3|2|3|3|3 options for 96k+4k dataset