Notes for use:** This document is the full report for this project. The models were run in the run_models.ipynb notebook. The exploratory data analysis (graphs for description of data) can be found in eda.ipynb and the analysis of the final results (graphs for numerical evaluation) from all models are in results_analysis.ipynb. All model related functions are stored in the project_models.py file and the model_functions.py contains helper functions for loading, setting configurations and plotting histories etc. Losses, accuracies and histories for all models can be found in this Drive folder: https://drive.google.com/drive/folders/1_X12r2eu7VW918xviQBQKaxg6o5XUl_J

Sentiment Analysis of Movie Reviews: A Comparison of Deep Learning Architectures and Embeddings

Abstract

In this paper, we attempt to analyze the sentiment of movie reviews from the Internet Movie Database (IMDb) using deep learning techniques to classify them as being either positive or negative. With the rise of social media and increasing digitalization, the need to detect sentiment in text accurately has never been more important. We compare multi-layer perceptrons (MLP), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), a Long-Short Term Memory (LSTM) model, and a Gated Recurrent Unit (GRU) model to determine which architectures are most effective for that task. We also look to vary the word embeddings used in each model, using combinations of the above models with one of either no embedding, a custom Keras embedding, and the pre-trained GloVe and spaCy embeddings. In addition to this, we try to optimize performance by employing batch normalization and Elastic Net regularization. We find that the MLP with Keras embedding and no regularization performs best, achieving an accuracy of 89.4%. Across embeddings, Keras-embedded models performed the best and across architectures GRUs were most effective. The effects of batch normalization and regularization are less clear cut, but it is likely that batch normalization is somewhat helpful.

Introduction

With the advent of the internet in general and social media in particular, the amount of text data available to businesses and researchers has increased exponentially. This data holds a huge amount of value, stemming from the fact that the opinions of those around us are key in influencing our own behavior. One powerful method of extracting this value from within the realm of Natural Language Processing (NLP) is sentiment analysis, one of the most popular areas of research within the NLP space in recent years. It entails "the computational study of people's opinions, sentiments, emotions, appraisals, and attitudes" (Zhang, Wang and Liu, 2018) via text mining in order to extract subjective information, such as whether the tone is positive or negative overall. The applications of sentiment analysis are therefore wide ranging, including brand monitoring on social media, the management of political campaigns, and the analysis of trends within financial markets.

In this paper, we seek to apply sentiment analysis techniques to movie reviews from the Internet Movie Database (IMDb), in order to classify them in terms of whether they are overall positive or negative. There are various machine learning techniques that can be used to achieve this goal, such as Support Vector Machines and Logistic Regression, however, we look to apply cutting-edge deep learning approaches. We conduct this classification using five separate architectures: a feedforward Multi-Layer Perceptron (MLP), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), a Bidirectional Long-Short Term Memory (LSTM) architecture, and finally a Bidirectional Gated Recurrent Unit (GRU) architecture. We then further tune these models by implementing two regularization techniques: Batch normalization and Elastic Net regularization.

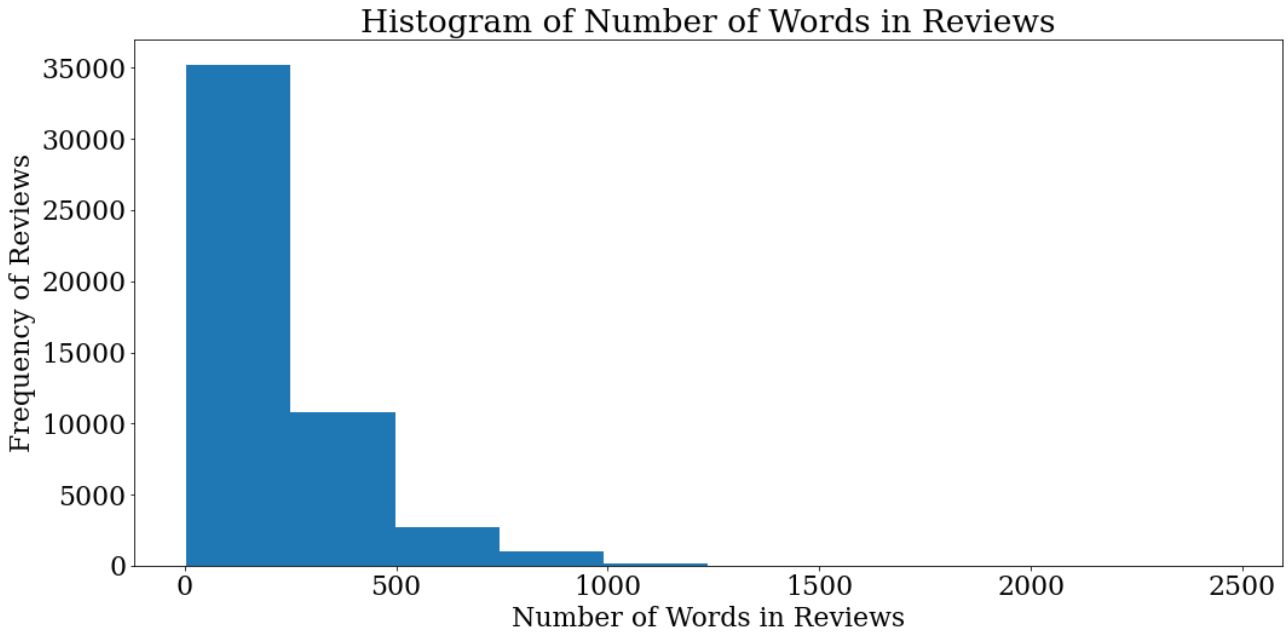
Within these architectures, we base our analyses strictly on the sequences of text constituting the reviews, due to the necessarily sequential nature of text data. This is because, unlike in other machine learning problems such as the prediction of house prices based on variables like tax rates or the existence of a nearby river as they appear in the Boston House Prices Dataset (Harrison & Rubinfeld, 1978), words (understood as variables) in a text structure are not independent of each other. For instance, in the sentence "this is a good ___", there are words which have a higher probability of appearing than others, simply based on the structure of the previous words. I.e., the words "movie" or "pizza" are more likely to appear than "translate" because the former are nouns and the latter is a verb. Consequently, words should not be assumed to be independent variables by the model that is trying to solve the task of understanding text for sentiment analysis. To use the text in our models, we use four different word embedding variations (no embedding, GloVe, Spacy, custom Keras) to see which yields the most accurate classifications.

In terms of results, out of the embeddings used, the Keras embedding achieved the greatest accuracy averaged across models. This implies that the use of an embedding that is not pre-trained, in comparison to SpaCy or GloVe, is important for learning the semantic context of the specific corpus used. Out of the model architectures, GRUs and LSTMs were the best performing – this is somewhat unsurprising given the need to process sequential data in this task, which suits these architectures in particular. Finally, the use of batch normalization or ElasticNet regularization did not significantly improve performance to

the same extent as the choice of architecture or embedding. In fact, those networks with only an ElasticNet layer but not batch normalization performed the worst on average. The top-10 performing models achieved a test accuracy of around 85%, with the best model achieving 89.4%

Description of Data

This project relies on data from the Large Movie Review Dataset (Maas et al., 2011). It is a collection of 50,000 reviews from the movie reviewing site IMDb. The reviews that have been included are highly polarized, meaning the reviews are either strongly positive or strongly negative in their sentiment. There are an exactly even number of positive and negative reviews. From this, a randomly selected 20% of reviews were held out to use as a test set. A further 20% of the remaining data was held out as a validation set during training.



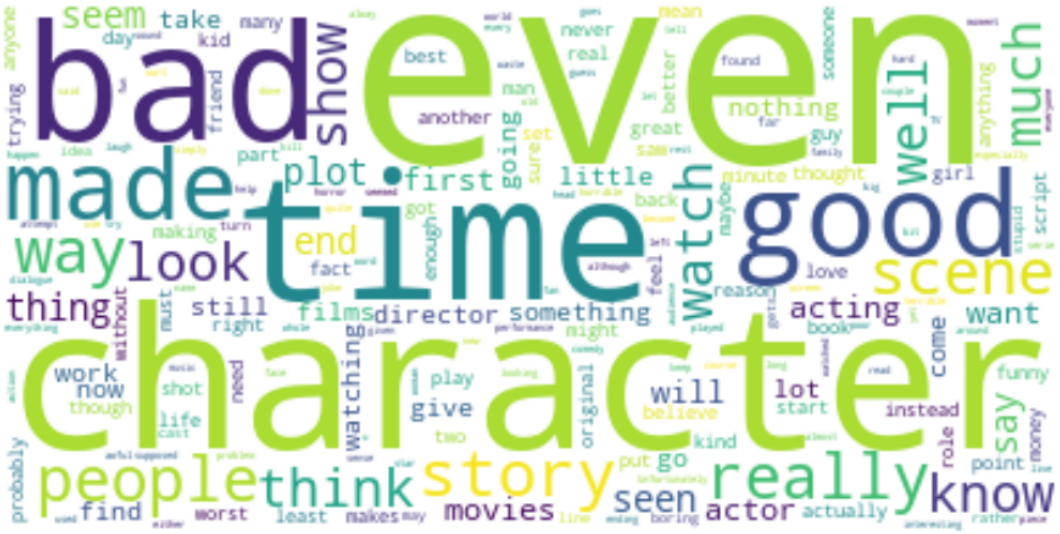
The reviews were processed to remove special characters and punctuation and converted to lowercase. Reviews were also filtered to be a maximum of 500 words long. As can be seen from the histogram, such a cut-off is high enough to cover the majority of reviews. Truncation for reviews over 500 words was used at the end of reviews rather than at the start, as intuitively one would think people may write their more important review points at the start.

A Keras Tokenizer object was used to learn a representation of the vocabulary in the dataset and then convert them into sequences of numbers. Pre-padding was used for sequences with length less than 500 to ensure they were all the same length. Pre-padding was employed as it has been found to greatly improve the performance of RNN-style models vs post-padding (Dwarampudi & Reddy, 2019)

An insight into what the most frequent words are (after filtering out stopwords) for positive and negative reviews can be seen in the two following word clouds.

Word Cloud of Positive Reviews

Word Cloud of Negative Reviews



Solution Methodology

General Approach

80 models were trained for this project, drawn from a grid/combinations of:

- **5 architectures:** MLP, CNN, RNN, Bi-LSTM and Bi-GRU
- **4 embeddings options:** None, GloVe, SpaCy, custom Keras
- **4 combinations of regularization and batch normalization:** None, only regularization, only batch normalization, both regularization and batch normalization

Additional architectures etc. could have been deployed but with each extra dimension added to the grid of possible models, the computational requirements increase considerably and finding an optimal solution becomes harder. This selection of models provided adequate scope to assess the problem while at the same time being feasible given limited computational resources.

In order to make the comparisons of models as valid as possible all features except for the options specified above were kept consistent across all models. The key components that were used across all models are discussed below, with a fuller discussion of each specific architecture given later on. Also, where possible, layer units were specified in powers of 2 as many practitioners believe this has advantages for the performance of processors (e.g. GPU) (Radiuk, 2017).

Loss Function: Binary Cross-Entropy

When training a model we seek to reduce the error it produces - the loss function tells us how to update our model's weights/parameters in order to produce better predictions and minimize our error.
Our problem is one of binary classification, so the binary cross-entropy function is the appropriate choice of loss function. Binary cross-entropy is the negative average of the log of corrected predicted probabilities. It calculates the average difference between the actual and predicted probability distributions for one of our two classes and can be specified as:

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

It is specified as a negative as we want to minimize loss.

Output Activation Function: Sigmoid

An activation function takes the summed, weighted input for a node and transforms it into the output for that input. Again, given that our problem is binary classification, the sigmoid function was used as the activation for the final layers/output layers in all the models. The sigmoid function takes the set of all real numbers as its domain and has a range of (0, 1) - meaning that the output for all of our models is guaranteed to be squashed to between 0 and 1, i.e. the target labels of our data.

It is defined as:

$$S(x) = \frac{1}{1 + e^{-}x}$$

Dense Layer Activation Function: ReLU

Rectified Linear Unit (ReLU) activation functions were used in the hidden Dense layers of all the models. ReLU is a piecewise linear function that will output the input directly if the input is positive, otherwise it outputs zero. This enables it to have linear behavior but at the same time have a non-linearity which allows it to learn more complicated data structures. They also avoid the easy saturation that affects other popular activation functions such as the sigmoid or hyperbolic tangent, whereby all very large values are forced to 1 and all very small values are forced to 0 or -1 respectively.
ReLUs have been shown to be quicker to train with and often better performing than certain other activation functions (e.g. Dahl, Sainath & Hinton, 2013).

It is defined as:

$$f(x) = max(0, x)$$

Optimization Algorithm: Adam

Adam, or adaptive moment estimation, is an optimisation algorithm that has been used in a variety of deep learning applications. It is computationally efficient, has little memory requirements and is well-suited for problems that are large in terms of parameters such as this one (Kingma and Ba, 2015). It is designed to incorporate the benefits of and build on two earlier optimizing algorithms, RMSProp and AdaGrad. Like AdaGrad it performs well with sparse gradients (useful in natural language tasks) but it is generally faster than AdaGrad. Like RMSProp it works well in non-stationary settings, but it adds in bias-correction and momentum (Ruder, 2017).

The algorithm works by updating exponential moving averages of the gradient and the squared gradient, using given hyperparameters to control the exponential decay rates of these moving averages. The moving averages are estimated using the mean and the uncentered variance of the gradient (ibid.).

The parameter update rule for Adam is thus specified as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Where $\hat{m}_t$ is the bias-corrected first moment estimate of the gradient, $\hat{v}_t$ is the bias-corrected second moment estimate, η is the step-size/learning rate and ϵ is a smoothing term that helps avoid division by zero.

A range of learning rates and exponential decay rates for 1st and 2nd moments could have been tried, but in the interests of maintaining valid model comparisons and feasible computational loads the default values specified by Kingma and Ba (2015) were used.

Kernel Initializers

Kernel initialization is important as without it models can be prone to exploding gradients (exponential growth in model parameters) or vanishing gradients (parameters may become 0 or too small to learn with well) during training.

Glorot Uniform kernel initialization was used in all applicable non-Dense layers as they've been shown to help performance in a range of situations. They initialize weights in such a way that the variance of the activations are the same across each layer, which can help with exploding/vanishing gradients (Glorot & Bengio, 2010).

He Uniform kernel initialization was used in all Dense layers as these have been shown to outperform Xavier/Glorot initializations in certain situations, such as when used in conjunction with ReLU activation functions (He et al., 2015). Glorot initializations assume that the activations are linear which is invalid for ReLU.

Callbacks: Early Stopping

Callbacks are objects that can be used to perform various actions on models at different stages of training. Early stopping is a callback that can be used during training that has been shown to reduce training times and prevent overfitting (e.g. Prechelt, 2012). It was used here to track the validation loss, i.e. the loss on the validation set of data within an epoch, and was given a patience of 25 epochs. This meant that if an improvement in the validation loss for the model was not detected within 25 epochs, training would be halted. The callback was set up so as to return the weights from the epoch in which the best validation loss was observed, rather than the final/25th epoch after it.

Regularization: Elastic Net

Elastic Net is a regularization technique that can be used to help prevent overfitting and improve the generalization of a model (Zou and Hastie, 2005). It applies penalties to large weights in the model while it is learning which helps to reduce its variance, at the cost of some bias. It uses a combination of $\ell 1$ (LASSO / sum of absolute values of coefficients) and $\ell 2$ (ridge regression / square root of the sum of the squared coefficients) penalties such that the loss function that is being minimized is no longer just binary cross-entropy, but binary cross-entropy and these penalty terms.

Elastic Net tries to combine the best parts of LASSO and ridge regression- the $\ell 1$ component performs automatic feature selection (reducing the coefficients for certain features to exactly zero), while at the same time the $\ell 2$ component encourages grouped selection and can help retain features that are highly correlated. LASSO on its own would arbitrarily pick one of the correlated features to keep and ridge on its own would not completely remove irrelevant features (Ogutu, Schulz-Streeck & Piepho, 2012). Intuitively, many words in negative or positive reviews are likely to be synonyms and so highly correlated and many words are likely to be irrelevant/noise and so could be canceled to zero - so Elastic Net would seem to be an appropriate choice of regularization. The default regularization factors of 0.01 were used for both the $\ell 1$ and $\ell 2$ components. It could have been useful to experiment with a range of values for these factors but given limited computational resources and the already high number of models to train, this was not feasible.

It is defined as below, where BCE is the binary cross-entropy loss defined earlier and w are the parameters we're trying to estimate:

$$\text{Penalized Loss} = BCE(w) + \lambda \|w\|_1 + \lambda \|w\|_2$$

Batch Normalization

Batch normalization is a method for standardizing inputs to a layer for each mini-batch, which allows us to stabilize the learning process thus reducing the number of training epochs required for a network. Hence, it allows us to significantly accelerate the training process. The need for batch normalization arises from the fact that, when training, the model is updated in each layer from output to input using estimates of error that assume weights in the previous layers are fixed. Therefore,

since all layers are changed during an update, the "target" for the update procedure is always moving. For instance, the weights of a layer are updated given an expectation that the previous layer outputs values with a given distribution. The distribution of each layer's inputs changes during training, as the parameters of the previous layers change. Consequently, this slows training down by requiring lower learning rates and more meticulous parameter initialisation. Ioffe & Szegedy (2015) refer to this "change in the distribution of internal nodes…in the course of training" as Internal Covariate Shift (ICS). In other words, ICS occurs when our model has learned some X to Y mapping, but the distribution of X changes – in this case, we would then need to retrain the model to accommodate for this. This is true even if the ground truth function mapping X to Y remains unchanged.

To combat this, we look to standardize our layer inputs. Batch normalization achieves this by rescaling the outputs of the prior layer, such that the input of the following layer has a mean of zero and a standard deviation of one, in line with a standard Gaussian distribution. Doing this means that the assumptions that each subsequent layer makes about the spread of the incoming inputs during weight updates do not, or at least are unlikely to, change. This therefore stabilizes and accelerates training.

There are also various other benefits to employing batch normalization. Firstly, it allows for the use of higher learning rates. In cases without batch normalization, high learning rates often result in exploding or vanishing gradients, in addition to getting stuck in local minima. However, after employing batch normalization, small changes in the parameters are prevented from increasing into larger changes which lead to suboptimal changes in activations in gradients. Secondly, it also has a beneficial effect on the gradient flow through the network, by reducing the dependence of gradients on the scale of the parameters or of their initial values, given the rescaling that occurs. This allows us to use much higher learning rates without the risk of divergence. Also, it acts as a regularization technique. It achieves this by adding noise to each hidden layer's activations – it does this both multiplicatively, given that it scales by the standard deviation, and additively, as it subtracts the mean. The downstream hidden units are therefore not too heavily reliant on any one prior hidden unit, in a similar way to Dropout, as per Srivastava et al. (2014). Batch normalization therefore has a slight regularization effect thus reducing the need for Dropout – hence we choose not to use Dropout in conjunction with batch normalization in our analysis.

In terms of successful applications of batch normalization in the literature, Ioffe & Szegedy (2015) apply batch normalization to the Inception network proposed by (Szegedy et al., 2014), trained on the ImageNet classification task. With an initial learning rate of 0.0015, the model achieves an accuracy of 72.2% after training steps. The network consists of a large number of convolutional and pooling layers (with ReLU activation function), with a softmax output layer to predict the image class (out of 1000 possible classes). By employing batch normalization alone, they match the accuracy achieved by Inception "in less than half the number of training steps." By increasing the learning rate by a factor of 5 (to 0.0075) and using a Sigmoid instead of a ReLU non-linear activation function, they reach the target 72.2% accuracy with 14 times fewer training steps. This demonstrates the effectiveness of batch normalization in accelerating the training process.

All of the above factors contribute to the inclusion of the option to use batch normalization in our models. In terms of implementation, we apply batch normalization after the hidden linear dense transformation but before the activation function, as per (Ioffe & Szegedy, 2015).

Multi-layer Perceptron

As stated in the general methodology, the goal of this project was to compare various architectures, embeddings and batch normalizations/regularizations in as fair a way as possible, in order to make the comparisons valid. With this in mind, the MLP outlined here was also used as the final layers in each other model.
In this way, each comparison can be thought of as a comparison between a baseline MLP and a baseline MLP plus an additional architecture/layer, e.g. the GRU models are the same as the MLP models except they have an additional GRU layer inserted between the embedding layer and the first hidden Dense/MLP layer.

The Dense/MLP layers are fully-connected, meaning each neuron is connected to each neuron of the preceding layer. A perceptron is just a linear classifier- it yields an output from real inputs by forming a linear combination with its input weights (which is then passed through a ReLU activation in our case).

$$\phi(w^\intercal x + b)$$

The use of multiple perceptrons in sequence (multi-layer) allows the model to better learn non-linear functions that a single perceptron may struggle with.

Each MLP model consisted of an optional embedding layer, followed by a hidden layer of 256 units and then an output layer- with options to toggle regularization and batch normalization. 256 units creates a feature space that can learn fairly complex functional forms but is not so large as to drastically increase the number of parameters and hence training times (particularly important for later models).

Convolutional Neural Network

As a development on the MLP, we look to use variations of a Convolutional Neural Network (CNN) in this task. CNNs are made up of an input layer, a convolutional layer, a pooling layer, and a fully-connected layer. Examples of their use in the literature for sentiment analysis include analysis of the Stanford Sentiment Treebank corpus by (Severyn and Moschitti, 2015), whilst (dos Santos and Gatti, 2014) use CNNs to analyze Twitter sentiment.

The convolutional layer determines the output of neurons which are connected to local regions of the input through the calculation of the scalar product between their weights and the region connected to the input volume. Hence, it is able to extract local features given its restriction of receptive fields of the hidden layers to be local. As a result, CNNs in particular have "a special spatially local correlation by enforcing a local connectivity pattern between neurons of adjacent layers" (Zhang, Wang and Liu, 2018). This characteristic is useful for classification in NLP problems in particular, given that we expect to find local clues regarding class membership, although these clues can appear in different places in the input. In the context of our task, a single key phrase within a review might imply whether it is positive or negative – if that is the case, we want to learn these sequences but do not care where exactly they appear in the document. The benefit of convolutional layers, in conjunction with pooling layers which perform downsampling thus reducing the number of parameters, is that they allow us to find these local indicators regardless of their position in the document.

Each CNN used in our analysis was constituted by an optional embedding layer, a one-dimensional convolutional layer with 256 filters and kernel of length 8, a one-dimensional maximum pooling layer with a pool size of 2x2. The output is then flattened and passed through a hidden dense layer with 256 units and ReLU activation function, until it is finally passed through a Dense output layer with a sigmoid activation function to yield a positive or negative classification.

Recurrent Neural Networks

Before the introduction of Recurrent neural networks, language had been modelled through the use of autoregressive models, where the probability of a word appearing at the present moment was conditioned by the words that appeared previously (for an extensive discussion of the topic, see Saul and Pereira (1997). I.e., if x_t is the current word that we want to predict, its probability of appearing after a sequence of words is $P(x_t/x_{t-1}, \ldots, x_1)$. If we only want to condition the current word on the past τ instead of on all previous observations, i.e. $P(x_t/x_{t-1}, \ldots, x_{t-\tau})$, that would be equivalent to a τ-order Markov model. This can also be modelled through Feedforward Neural Networks (or other ML models). The choice of τ brings some issues. If τ is too small, we will not be able to capture long-term dependencies between words. On the other hand, as τ grows, the total number of word combinations that must be stored also soars. E.g., if we had a vocabulary set V, we would need to save the frequencies of each word or word combination in this vocabulary set, creating $|V|^{\tau+1}$ variables. Hence, these models get unwieldy fast. However, recurrent neural networks solve both problems through a different modelling approach.

Unlike the formulation of autoregressive models, recurrent neural networks do not make explicit use of all the previous word frequencies. Instead, they rely on a **hidden state** that stores the relevant information contained in the sequence. In mathematical terms, this hidden state at the moment t is written as h_t. Hence, the probability of a certain word appearing after the previous sequence at moment t is defined as $P(x_t/h_{t-1})$. On the other hand, this hidden state at any time t is a function of the previous state and the input at the same moment, i.e., $h_t = f(x_t, h_{t-1})$. As a result, RNNs use the recurrent computation of its hidden state to preserve long-term information in a sequence. Nevertheless, in order to avoid storing everything that happened previously into the hidden state, the model will adjust the weights of the function that models $f(x_t, h_{t-1})$ during training to preserve only relevant information.

For a neural network implementation, imagine we have a minibatch $X_t \in \mathbb{R}^{n \times d}$, with n being the batch size and d the vocabulary size. The input hidden state that we want to calculate is $H_t \in \mathbb{R}^{n \times h}$, being h the number of hidden units that we specify for the hidden layer. This is modelled as:

$$H_t = a(X_t W_{xh} + H_{t-1} W_{hh} + b_h),$$

where $W_{xh} \in \mathbb{R}^{d \times h}$, $W_{hh} \in \mathbb{R}^{h \times h}$, $b_h \in \mathbb{R}^{1 \times h}$, and a is an activation function; usually (and also in our case, hyperbolic tangent).

The tanh (hyperbolic tangent) activation function has the following shape:

$$tanh(x) = \frac{e^x - e^- x}{e^x + e^- x}$$

Finally, the output $O_t \in \mathbb{R}^{n \times q}$, where q is the relevant number of variables that we want to predict (i.e., q = 1 in binary classification or regression, > 2 for multi-class classification) using the RNN is defined as:

$$O_t = H_t W_{hq} + b_q,$$

where $W_{hq} \in \mathbb{R}^{h \times q}$, and $b_q \in \mathbb{R}^{1 \times q}$. As a result, W_{xh}, W_{hh}, W_{hq}, b_h, and b_q are all trainable weights of the recurrent neural network, and they are used to preserve sequential information.

As a side note, because of the long computational chain of backpropagation through time in sequence data, RNNs usually suffer from either vanishing or exploding gradients. The latter can be controlled through gradient clipping (setting a maximum to the norm of the gradient so that weight updates do not grow exponentially), but the former, which happens more frequently (Cho et al., 2014) is better handled through newer architectures that we will explain later, namely LSTMs and GRUs.

In our case, we were not interested in modelling language directly, but analysing whether the text contained in movie reviews should be perceived as either positive or negative. Hence, we had to use an additional Dense layer for binary classification. Furthermore, given that in a movie review (and many other sentiment analysis tasks) the whole text sequence is available, we can apply bidirectional RNNs, which process the input in both directions. This has also been used in the literature (Ding et al., 2018) for text classification tasks, since using both directions of the sequence can help with semantic understanding. Finally, given that here we are trying to compare different embeddings and generalization tweaks (batch normalization and regularization), we reproduce a similar architecture. As a summary, the RNN architecture will be:

Embedding -> Bi-directional RNN with 32 * 2 hidden units -> Fully-connected layer with 256 hidden units and ReLu activation function -> Output layer with sigmoid activation function for binary classification.

Long Short-term Memory

Even though recurrent neural networks are meant to store the relevant information of the previous items in a sequence, the long computational chain of backpropagation updates might face the problem of vanishing or exploding gradients. As mentioned above, exploding gradients are dealt with by bounding the gradient to a maximum norm size. Nevertheless, after taking this into account, vanishing gradients are still a source of concern. One proposed solution for RNNs is to truncate part of the computation with either a fixed window size or randomization. However, here we will dive deeper into the second and most popular solution, which relies on creating trainable gates that separately control the flow of short-term and long-term information. This is implemented in both LSTMs and GRUs.

LSTMs were the first fix, proposed by (Hochreiter and Schmidhuber, 1997). Since then, they have also been used for text classification. One example is text classification of customer opinions and political leanings (Rao & Spasojevic, 2016), whereas other authors have applied it to classification of source code documents according to a programming language (Reyes et al., 2016). On the whole, it is a widely-used model for text data.

The main idea of LSTM consists in creating an additional memory cell that preserves long-term information, and whose influence on the hidden state is controlled through three gates. One of the gates is meant to read the information in the memory cell and pass it onto the hidden state, this is called the *output gate*. An additional gate is used to decide which information of the current X_t goes into the current memory cell, this one is called *input gate*. Finally, there is a gate to control when to reset the memory cell from long-term information that is no longer useful; it is called a *forget gate*.

To begin with, instead of making H_t depend directly on H_{t-1} and X_t through one affine transformation plus activation function, there are several intermediate steps. To start with, gates are introduced to control the flow of information:

$$\text{Forget gate: } F_t = \sigma(X_t W_{xf} + H_{t-1} W_{hf} + b_f)$$
$$\text{Input gate: } I_t = \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i)$$
$$\text{Output gate: } O_t = \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o),$$

where σ is the sigmoid function, because all elements in the gates will range between 0 and 1 for their following computation. Furthermore, the dimensions of all the gates are $n \times h$. Hence, $W_{xf}, W_{xi}, W_{xo} \in \mathbb{R}^{d \times h}$; $W_{hf}, W_{hi}, W_{ho} \in \mathbb{R}^{h \times h}$; and the biases $b_f, b_i, b_o \in \mathbb{R}^{1 \times h}$.

As shown, these gates have learnable weights and biases that will be constant for any time step. The aim of the first two is to change the information that goes into the current memory cell. First, the forget gate controls how much of the previous memory cell is preserved or, in other words, is able to reset the state of the previous cell. On the other hand, the input gate

interacts with the current candidate memory cell to create the memory cell. Mathematically, the candidate memory cell is created through an NN transformation of the current minibatch X_t and the previous hidden state H_{t-1}. I.e., the candidate memory cell is defined as:

$$\widetilde{C}_t = tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c),$$

where $\widetilde{C}_t \in \mathbb{R}^{n \times h}$.

Hence, the final memory cell that will be passed to the next time step is:

$$C_t = F_t \odot C_{t-1} + I_t \odot \widetilde{C}_t$$

Given that both F_t and I_t are matrices whose elements range between 0 and 1, if $F_t \approx 1$, and $I_t \approx 0$, then C_{t-1} will pass between time steps almost unaltered. Hence, this fact helps to avoid the vanishing gradient problem, and allows the network to capture long-term dependencies.

Nevertheless, there is one more final step. Even though now we have a final memory cell C_t that will pass to the following time step, we still have not altered the current hidden state H_t. This is the job of the last gate, O_t, which controls the influence of the cell on the hidden state. The operation is:

$$H_t = O_t \odot tanh(C_t), tanh(C_t) \in (-1, 1)^{n \times h}$$

Finally, if $O_t \approx 1$, then all the information in the memory cell is passed onto the hidden state.

In conclusion, the operation $H_t = f(X_t, H_{t-1})$ is now less straightforward in an LSTM than in a simple RNN, since it passes through several gates and multiple operations (which could be considered mini-feedforward neural networks because the transformations that they apply are equivalent). But, in essence, it depends on both of them (plus the memory cell, although this also stems from the first X_0, H_0).

The implementation used is as similar as possible to the RNN, since we wanted comparable models:
Embedding -> Bidirectional LSTM with 32 * 2 units -> Fully-connected layer with 256 hidden units and ReLu activation function -> Output dense layer with 1 unit and sigmoid activation function for binary classification

Gated Recurrent Units

As explained before, gated recurrent neural networks were proposed as a solution to vanishing gradients. The first of them were LSTMs. But more recently, Gated Recurrent Units were proposed as an alternative architecture (Cho et al., 2014) that consists in a similar use of gating to control when to update the hidden state. They have also been extensively used in text classification (Tang et al., 2015).

First we will outline the new architecture theory, and show the main similarities and differences with LSTMs. To begin with, instead of having three gates as in LSTM, there are only two of them, namely:

$$\text{Reset gate: } R_t = \sigma(X_t W_{xr} + H_{t-1} W_{hr} + b_r) \in [0,1]^{n \times h}$$
$$\text{Update gate: } Z_t = \sigma(X_t W_{xz} + H_{t-1} W_{hz} + b_z) \in [0,1]^{n \times h}$$

In this model architecture there are no memory cells. Instead, the gates control how much of the previous hidden state H_{t-1} is passed to the following one H_t. The first operation involves the reset gate, which calculates how much of the previous hidden state goes into a candidate hidden state. This is calculated as:

$$\widetilde{H_t} = tanh(X_t W_{xh} + (R_t \odot H_{t-1}) W_{hh} + b_h)$$

Hence, if the reset gate equals 1, the candidate hidden state will be calculated in exactly the same way as the hidden state in a simple RNN. In other words, the reset gate allows the model to remember or discard the information of the previous state.

Once we have our current candidate hidden state, we can just calculate the current hidden state by using the update gate as follows:

$$H_t = Z_t \odot H_{t-1} + (1 - Z_t) \odot \widetilde{H_t}$$

As a result, if the update gate was close to 1, the current state would be almost a copy of the previous state. Hence, the reset gate would be able to control short-term relationships between variables; whereas the update gate can pass long-term dependencies.

Similarities and differences of LSTM and GRU: First, let us make the equivalences between the gates of both neural networks explicit. On the one hand, both the forget gate in LSTM, and the update gate in GRU help important features that appeared long ago to remain in the modelling sequence. On the other, the input gate in LSTM and the reset gate in GRU control the amount of new information that goes into a candidate state (either the memory cell or the hidden state).

Regarding their similarities, as outlined in Cho et al. (2014), both LSTMs and GRUs rely on storing some existing information through gates and adding the new minibatch of observations upon it. This allows to, first, remember a specific important feature that appeared long ago in the sequence and, second and more important, allow the error to backpropagate more easily, so that the vanishing gradients problem is dealt with. However, they are different as to how the memory content is exposed. In the LSTM unit, the output gate controls how much of the memory cell can be passed onto the hidden state. On the contrary, in the GRU the full candidate hidden state is exposed, without any gate controlling directly its effect (since the update gate only takes into account the copying of the previous state, but does not only control the candidate hidden state influence).

In spite of the changes between their architectures, in the paper of Cho et al. (2014) there were no clear empirical advantages of using any of them for speech recognition. Results were similar in Bahdanau et al., (2016), where there were no clear advantages between LSTMs and GRUs for machine translation. Nevertheless, the results are way better than those of simple RNNs.

The implementation is again similar to RNN's, namely:
Embedding -> Bidirectional GRU with 32 * 2 hidden units -> Fully-connected hidden dense layer with 256 units and ReLu activation function -> Dense layer with 1 unit and sigmoid activation function for binary classification.

GloVe Embedding

The Global Vectors (GloVe) Word Representation embedding is a "specific weighted least squares model that trains on global word-word co-occurrence counts" (Pennington, Socher and Manning, 2014). In comparison to other word embeddings, such as Word2vec, GloVe does not just rely on local statistics in terms of the local context information of words, rather it should incorporate global statistics in terms of word co-occurrence to obtain word vectors. Word2vec relies only on the words surrounding a given word that is to be learned. For instance, in the sentence "The cat sat on the mat", Word2vec would not consider whether the use of "the" in this case is special or necessary in this context or if it was simply a stopword. In contrast, the core of the argument for GloVe is that instead of training a word embedding simply on the probability of a given word occurring, we should instead be considering the ratios of co-occurrence probabilities. In this way it captures both global statistics and local statistics of a corpus, in order to come up with word vectors. Suppose we have a corpus of V words – the co-occurrence matrix will be a V x V matrix, where the ith row of the jth column of X, X_{ij}, denotes the number of times word i has co-occurred with word j. Let $P_{ij} = P(j|i) = X_{ij}/X_i$ – in other words, the probability that word j appears in the context of word i. To derive a metric measuring semantic similarity, if we consider word k, it is P_{ik}/P_{jk} that indicates similarity. For example, suppose i = "ice", j= "steam", and k = "solid". For words related to "ice" but not "steam", we expect the ratio to be large, for example, in the case of "solid". For words related to "steam" but not "ice", the ratio would be small. Finally, for words that are either related to both "ice" and "steam" or not related to either, the ratio is close to one.

SpaCy Embedding

An alternative pre-trained embedding we use in our analysis is the spaCy embedding. The spaCy pipeline first tokenizes the text object to yield a Doc object. This object then goes through various processes including tagging to assign part-of-speech tags, a parser to assign dependency labels which jointly learns sentence segmentation and labeled dependency parsing, and entity recognition to detect and label named entities. In our analysis we use the en_core_web_sm package, the small English pipeline offered by spaCy trained on written web text (blogs, news, comments), including vocabulary, syntax and entities, thus making it appropriate for use in this task. We iterate through our vocabulary, mapping words that are present in our corpus to their location in the spacy embedding. Finally, we load the embedding matrix as the weights matrix for the embedding layer in the relevant models.

Keras Embedding

Any Natural Language Processing task relies on proper modeling of text data. Nevertheless, to find the best way to help a machine learn how to understand text we must go into the details of words and their meaning. To begin with, the traditional approach to feed words into a model is the bag-of-words (BoW) approach. To do this, a vocabulary is created according to the frequencies of the words, so that each word has its own index in a lookup table. Afterwards, when we face a new word, this will be represented by a one-hot encoded vector, where all entries are zero except for the one that corresponds to this index. Hence, when we feed a word into our model, a weight is associated with its entry. Nevertheless, this approach lacks semantic context, since all pairs of vectors are orthogonal, so their distance is the same independently of their meaning. However, we would like our model to know that "stadium" is closer to "soccer" than to "car", or that "cucumber" has a smaller distance to "kitchen" than to "brick".

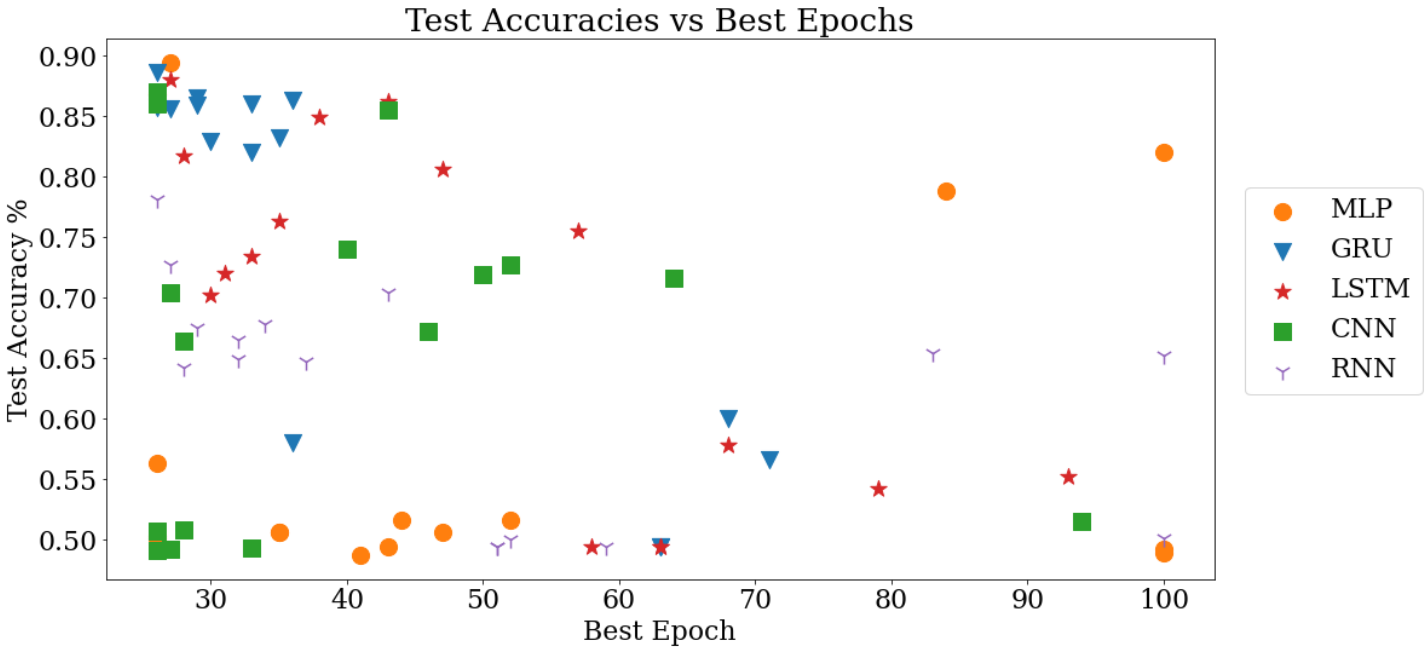
Word embeddings provide the solution to this problem. Let us explain it with an example. Imagine we have a vocabulary of 10,000 words. In a bag-of-words approach, we would end up with 10,000 one-hot encoded vectors of dimension $10,000 \times 1$. With word embeddings, we decide a dimensionality for each word to show its context. For instance, imagine that we choose an output dimension of 256. Then, the embedding would learn a weight matrix (through training) of $10,000 \times 256$. This weight matrix would work like a lookup table, so that if we had an observation with the index 156, then the vector that corresponds to row 156 would be extracted and used to represent it (after transposing it). Hence, we would represent each word through a 256×1 vector. Unlike the Bag-of-words approach, here the vector is not sparse. While learning the weights, it tries to minimize the distance (such as cosine distance) between words that are similar to each other. Hence, the distance between "cucumber" and "kitchen" will be smaller than that of "cucumber" and "brick".
As a result, the Keras embedding will try to learn these representations starting from random weights, so that the embedding layer learns the context of words for a particular dataset.

Numerical Evaluation

Top 10 Performing Models By Accuracy

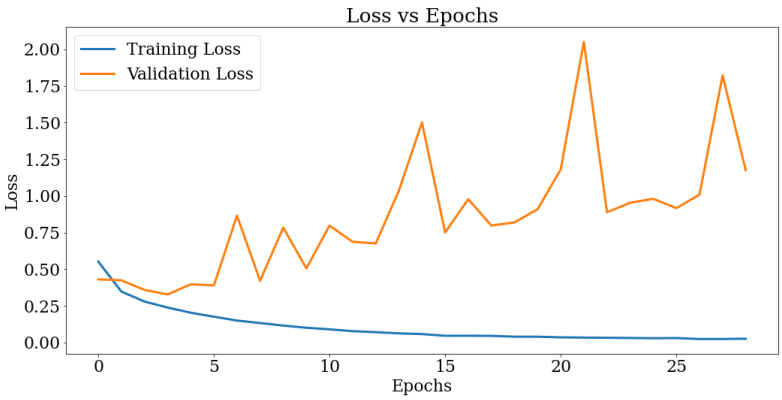
Architecture	Embedding	Regularized	Batch Normalized	Accuracy	ROC AUC	F1 Score
MLP	Keras	FALSE	FALSE	89.41%	95.95%	89.22%
GRU	Keras	FALSE	FALSE	88.67%	95.14%	88.42%
LSTM	Keras	TRUE	TRUE	88.00%	95.08%	87.51%
CNN	Keras	FALSE	FALSE	87.02%	94.32%	87.41%
MLP	Keras	FALSE	TRUE	86.85%	94.18%	87.02%
GRU	GloVe	FALSE	TRUE	86.52%	94.14%	86.26%
GRU	GloVe	TRUE	FALSE	86.30%	93.75%	86.45%
LSTM	Keras	TRUE	FALSE	86.23%	93.06%	86.55%
GRU	GloVe	TRUE	TRUE	86.05%	93.89%	85.90%
CNN	Keras	FALSE	TRUE	86.02%	93.39%	86.00%

A range of model architectures seemed to be able to solve this problem quite well, with several models reaching 85%+ accuracy. The order of best performing models did not change considerably depending on what metrics they were ranked by (likely reflecting the high class balance in the data), but they have been sorted by accuracy anyway. The best model that was identified was a Keras-embedded MLP, without regularization or batch normalization, which achieved an accuracy of 89.41%. This is slightly stronger than the best result for accuracy that the authors of the dataset (Maas et al., 2011) were able to achieve - 88.89% using a support vector machine (SVM). It also appears to have been a more successful approach than Timmaraju and Khanna (2015) who use a combination of recurrent, recursive neural networks and SVMs, along with a word2vec embedding, and only achieve a best accuracy of 86.5%.

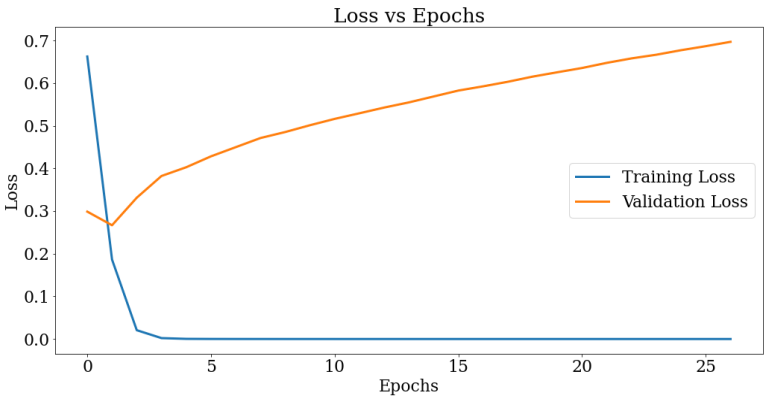


A more detailed picture of the performances of the 80 models can be seen in the graph above, which depicts the test accuracy for each model against the number of epochs they were trained for. There is a concentration of points in the top left corner, reflecting a number of models that had short training times but achieved high test accuracy. All of the top 10 models contained either a Keras or GloVe embedding layer which could perhaps explain these shorter training times. The models need only a few epochs to learn (Keras) or adapt to (GloVe) the embeddings and then as soon as this was accomplished, they began to overfit.

The loss curves for the best performing GloVe- and Keras-embedded models are displayed below and would seem to support this conclusion. Validation loss for the GRU reaches a minimum within a few epochs as it adapts to the GloVe embedding and the MLP reaches its minimum immediately as it learns the custom embedding for this data.



GRU GloVe

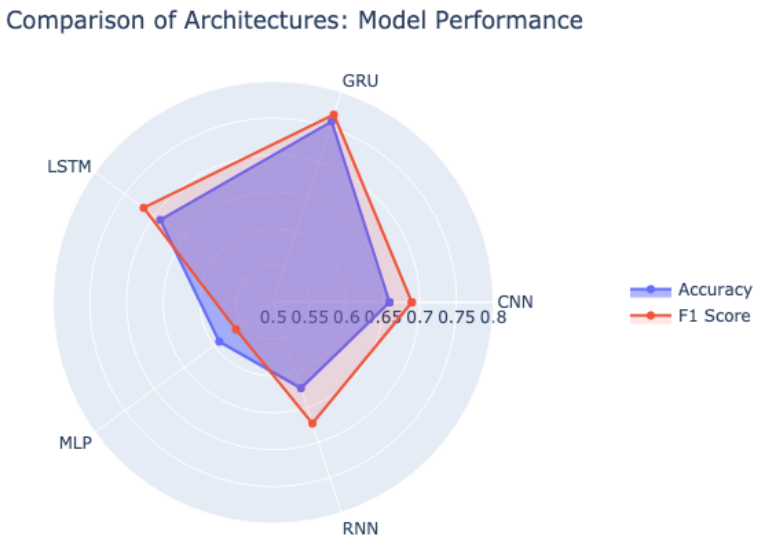


MLP Keras

Mean Performance by Architecture

Architecture	Accuracy	ROC AUC	F1 Score
GRU	75.78%	81.84%	76.77%
LSTM	69.02%	74.23%	71.82%
CNN	65.86%	72.04%	68.90%
RNN	62.26%	67.23%	67.29%
MLP	59.08%	62.34%	56.25%

The GRU designs of models were the strongest performing on average, followed by the LSTMs. Given the sequential nature of the data, this result would seem to make sense.

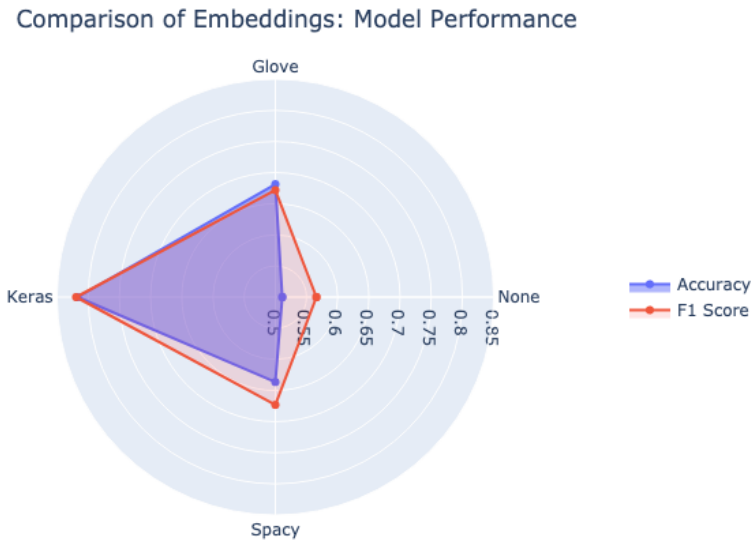


It is interesting to note that the MLP models were on average the weakest- despite one of them being the overall top performing model. This is perhaps a testament to the power of the Keras embedding (which it used) rather than because it was an MLP model.

It would have been useful to train the series of models a second time and see if the Keras-embedded MLP still performed the best, or if it was just a fluke. However, given the computational requirements this was not feasible.

Mean Performance by Embedding

Embedding	Accuracy	ROC AUC	F1 Score
Keras	81.72%	89.58%	81.94%
GloVe	68.14%	73.96%	67.18%
SpaCy	63.64%	69.19%	67.31%
None	51.11%	52.09%	56.59%



The Keras embedding was by far the strongest performing. The 256 embedding dimensions seem to have been enough to represent the text data reasonably well. However, given that it was trainable whereas SpaCy and GloVe embeddings were not (and that they were both trained on much larger corpora) it is likely that models using it would generalize to new reviews data less well.

The models without any embedding layer were not able to sufficiently learn the complexities of the data and so on average, were not much better than randomly picking half to be positive reviews and half to be negative.

Mean Performance by Regularization/Batch Normalization

Regularized	Batch Normalized	Accuracy	ROC AUC	F1 Score
TRUE	TRUE	68.37%	74.52%	68.51%
FALSE	TRUE	68.18%	74.69%	70.55%
FALSE	FALSE	66.81%	71.29%	64.70%
TRUE	FALSE	62.24%	65.64%	69.07%

In terms of accuracy, the combination of batch normalization and regularization resulted in the strongest models. However, in terms of F1 and ROC AUC the exclusive use of batch normalization gave the strongest models. Exclusive regularization appears to have been detrimental. Batch normalization is helpful to solving this problem and regularization, if it is helpful, is only so when used in conjunction with batch normalization.

Conclusion

The main aim of this project was to compare the performance on sentiment analysis of movie reviews of different neural network architectures (MLP, CNNs, Simple RNNs, LSTMs and GRUs). These could be combined with the use of four types of embeddings (out of None, Keras, GloVe, SpaCy) and layers of Batch Normalization or ElasticNet regularization could also be added. Several interesting results were found. First, models with Keras embeddings achieved the highest average accuracies compared with any other kind of embedding. Furthermore, the choice of embedding seemed to have the biggest impact on a model's performance - more than even the choice of architecture. This shows the importance of feeding semantic context to neural networks, and probably that the text nuances in movie reviews might be different from those in other smaller embeddings trained on news, such as SpaCy. Second, GRUs and LSTMs attained the highest average accuracies among all general model architectures, which was expected given that the task at hand involved processing sequential data. Finally, the choice of batch normalization or ElasticNet regularization was not as fundamental to model performance as the type of architecture or embedding used. However, networks with only regularization and no batch normalization had the worst performance of all on average. In the end, our top-10 models reached a test accuracy of 85+%, with a top model of 89.4%. Further experimentation could include the use of Keras embeddings initialized with weights of pre-trained embeddings on movie reviews, or tweaks to the model architecture, such as a higher number of hidden units in the hidden state of LSTM and GRU layers. It could also be interesting to see how well these models are able to generalize to other binary sentiment analysis tasks of other review datasets.

Individual Contributions

Aashrit: Design and implementation of CNN. Solution methodologies for batch normalization, CNN, GloVe and SpaCy. Write-up of abstract and problem formulation / introduction.

Alberto: Design and implementation of RNN and Bi-LSTM, data-loading functions. Solution methodologies for RNN, Bi-LSTM, Bi-GRU and Keras embedding. Write-up of conclusion.

Louis: Design and implementation of MLP, Bi-GRU, embedding layers and general project functions. Solution methodologies for the general aspects and MLP. Write-up of numerical evaluation and description of data.

References

Bahdanau, D., Cho, K. & Bengio, Y. (2016) Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv:1409.0473 [cs, stat]*. http://arxiv.org/abs/1409.0473.

Cho, K., van Merrienboer, B., Bahdanau, D. & Bengio, Y. (2014) On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *arXiv:1409.1259 [cs, stat]*. http://arxiv.org/abs/1409.1259

Dahl, G.E., Sainath, T.N. & Hinton, G.E. (2013) Improving deep neural networks for LVCSR using rectified linear units and dropout. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. May 2013 pp. 8609–8613. doi:10.1109/ICASSP.2013.6639346.

Datta, L. (2020) A Survey on Activation Functions and their relation with Xavier and He Normal Initialization. *arXiv:2004.06632 [cs]*. http://arxiv.org/abs/2004.06632.

Ding, Z., Xia, R., Yu, J., Li, X. & Yang, J. (2018) Densely Connected Bidirectional LSTM with Applications to Sentence Classification. In: *NLPCC*. 2018 p. doi:10.1007/978-3-319-99501-4_24.

Dos Santos, C., & Gatti, M. (2014). Deep convolutional neural networks for sentiment analysis of short texts. In Proceedings of COLING 2014, the 25th international conference on computational linguistics: technical papers (pp. 69-78).

Dwarampudi, M. & Reddy, N.V.S. (2019) Effects of padding on LSTMs and CNNs. *arXiv:1903.07288 [cs, stat]*. http://arxiv.org/abs/1903.07288.

Glorot, X. & Bengio, Y. (2010) Understanding the difficulty of training deep feedforward neural networks. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 31 March 2010 JMLR Workshop and Conference Proceedings. pp. 249–256. https://proceedings.mlr.press/v9/glorot10a.html.

He, K., Zhang, X., Ren, S. & Sun, J. (2015) Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv:1502.01852 [cs]*. http://arxiv.org/abs/1502.01852.

Hochreiter, S. & Schmidhuber, J. (1997) Long Short-Term Memory. *Neural Computation*. 9 (8), 1735–1780. doi:10.1162/neco.1997.9.8.1735.

Ioffe, S., & Szegedy, C. (2015, June). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International conference on machine learning (pp. 448-456). PMLR.

Kingma, D.P. & Ba, J. (2017) Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*. http://arxiv.org/abs/1412.6980.

Maas, A.L., Daly, R.E., Pham, P.T., Huang, D., Ng, A.Y. & Potts, C. (2011) Learning Word Vectors for Sentiment Analysis. *9. In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA. Association for Computational Linguistics. https://www.aclweb.org/anthology/P11-1015

Ogutu, J.O., Schulz-Streeck, T. & Piepho, H.-P. (2012) Genomic selection using regularized linear regression models: ridge regression, lasso, elastic net and their extensions. *BMC Proceedings*. 6 (2), S10. doi:10.1186/1753-6561-6-S2-S10.

Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation. *In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 1532-1543).

Prechelt, L. (2012). Early Stopping — But When?. In: Montavon, G., Orr, G.B., Müller, KR. (eds) Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science, vol 7700. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-35289-8_5

Radiuk, P.M. (2017) Impact of Training Set Batch Size on the Performance of Convolutional Neural Networks for Diverse Datasets. *Information Technology and Management Science*. 20 (1). doi:10.1515/itms-2017-0003.

Rao, A. & Spasojevic, N. (2016) Actionable and Political Text Classification using Word Embeddings and LSTM. *arXiv:1607.02501 [cs]*. http://arxiv.org/abs/1607.02501

Reyes, J., Ramírez, D. & Paciello, J. (2016) Automatic Classification of Source Code Archives by Programming Language: A Deep Learning Approach. *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*. doi:10.1109/CSCI.2016.0103.

Ruder, S. (2017) An overview of gradient descent optimization algorithms. *arXiv:1609.04747 [cs]*. http://arxiv.org/abs/1609.04747.

Saul, L. & Pereira, F. (1997) Aggregate and mixed-order Markov models for statistical language processing.

arXiv:cmp-lg/9706007. http://arxiv.org/abs/cmp-lg/9706007

Severyn, A., & Moschitti, A. (2015, August). Twitter sentiment analysis with deep convolutional neural networks. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval* (pp. 959-962).

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research, 15(1)*, 1929-1958.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. *In Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

Tang, D., Qin, B. & Liu, T. (2015) Document Modeling with Gated Recurrent Neural Network for Sentiment Classification. In: *EMNLP*. 2015 p. doi:10.18653/v1/D15-1167.

Tibshirani, R. (1996) Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*. 58 (1), 267–288.

Timmaraju, A., & Khanna, V. (2015). Sentiment analysis on movie reviews using recursive and recurrent neural network architectures. Semantic Scholar, 1-5.

Zhang, L., Wang, S., & Liu, B. (2018). Deep learning for sentiment analysis: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 8*(4), e1253.

Zou, H. & Hastie, T. (2005) Regularization and Variable Selection via the Elastic Net. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*. 67 (2), 301–320.