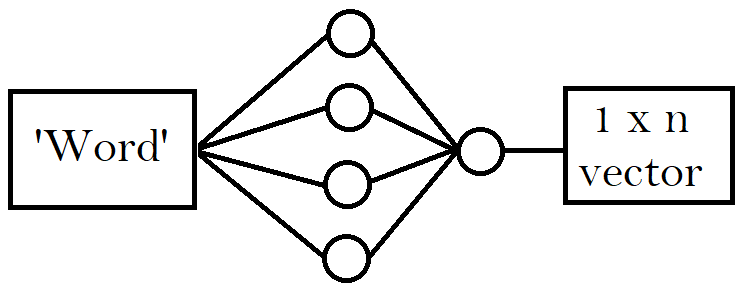
# Word Embeddings:

To train deep neural networks, we need a way to encode the words in the corpus in a representation that the ANN “understand” and can extract as much meaningful information from. One approach would be one hot encoding but given our dataset, that would be a very memory inefficient approach. Another alternative would be to train a shallow neural network to map each word to a vector space. The idea is that words with similar context will be mapped to vectors with similar value. The computational cost required to train such transformation is immense as, due to the time constraints, we only train the model for 20 epochs on a fraction of our dataset. For this project, we believe that these constraints should not invalidate any preliminary results from this analysis. We plan to make our code more efficient and train for longer epochs over the entire dataset in future.

We utilize PyTorch for creating a word embedding model. The first step was to give each word some context for the model to learn. This was done by supplying each word with 2 words preceding it and 2 words in its succession as context. We shall refer to this tuple as the word-context ngram. The next step was to assign each word a unique key/index to lookup its vector implementation after training the embedding model. We created a NGramLanguageModular model as described in the PyTorch documentation to generate our word embeddings.

After training this model with the corpus subset, we created a few helper functions that translate the corpus information to our models in a readable vector format. This was done by initializing a torch tensor with zero values and shape (num\_docs, vector\_length, 5000). Here 5000 is an arbitrary number to fix the size of our training sample for simplicity. This length was chosen by dividing the total words in the subset by the number of documents and rounding to the nearest 1000. As one might expect, the actual document length might be lower or higher than this, hence the zero initialization. The words were then converted to vector formats using the trained embeddings and copied over to this tensor. A few more helper functions then restructured this tensor for the models they were used in.

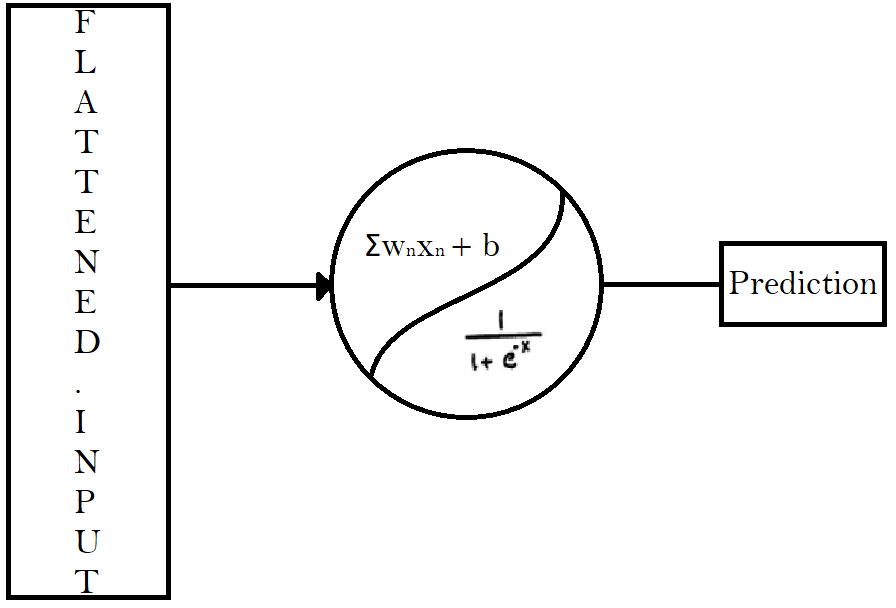
Logic:



## \*\* Note that the following models can easily be adapted to multi class classification by dropping sigmoid at the end and replacing BCE Loss with Cross Entropy Loss.

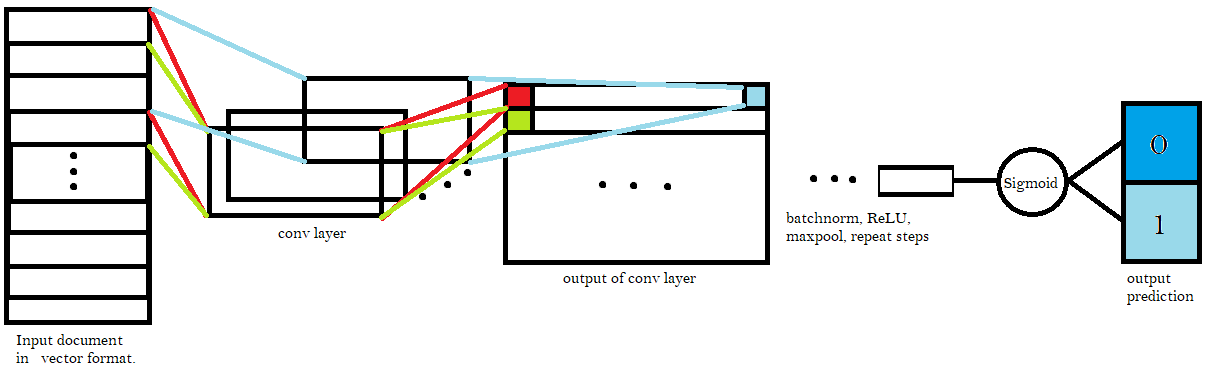
# Perceptron:

The perceptron is probably the simplest Feedforward neural network. It was set up as a benchmark model but gave decent results in the prototyping phase, so we decided to include it in this report as a benchmark to compare our model with. Our Perceptron consists of a singular “neuron” that takes the input per document as a flattened 1d array and uses sigmoid nonlinearity to predict a classification for the document. The model was coded to have a train, evaluate, save, and load functions to reduce complexity for the end user. The model utilizes ADAM optimization and BCE Loss (Log loss for binary classification) to update its parameters during backpropagation.



# CNN model:

The CNN (Convolutional Neural Network) model utilizes concepts from computer vison and applies them to the word vector document implementation. By using filters and windows, we can drastically reduce computational cost and parameter size while achieving similar or better performance as traditional dense neural nets. CNNs are a class of sparce networks since they share parameters per learnable window. The model architecture can be seen in the figure below (fig 1.2…). The model was coded to have a train, evaluate, save, and load functions to reduce complexity for the end user. The model utilizes ADAM optimization and BCE Loss (Log loss for binary classification) to update its parameters during backpropagation.



# LSTM model: (SUS)

LSTM model implements PyTorch’s provided LSTM class. Each LSTM module has 1 hidden layer with 3 nodes. Before passing the data to the LSTM layer during training, we drop 20% of the data randomly which translates to a probability of 0.8 for each input word to be passed through. This is known as dropout and in theory with large datasets, it should give better performance by reducing overfitting. The final hidden state of the LSTM model is passed to a linear to sigmoid sequence that outputs the predicted binary classification. The model architecture can be seen in the figure below (fig 1.2…). The model was coded to have a train, evaluate, save, and load functions to reduce complexity for the end user. The model utilizes ADAM optimization and BCE Loss (Log loss for binary classification) to update its parameters during backpropagation.

