ABSTRACT :

In today's world communication is done on a vast area along with the growth of the Internet. Communication and human interaction are carried out using social sites, applications etc. in textual form. Emotion plays a vital role in human day-to-day life conveying their feelings and situation in their life. Emotions can be expressed using speech, gestures facial expression, or text. Emotion Detection from textual source can be done using concepts of Natural Language Processing.(Add algo's, techniques, dataset etc. used)

Convolutional and Recurrent Neural Networks have shown striking results in the unconditional and conditional task but have not been worked upon as much as facial and emotional speech recognition. In this report, we implement several techniques to successfully obtain a model for emotion detection and recognition via text. After implementing the standard RNNs, they are improved by implementing FastText giving similar results in almost half the time. Thereafter we lay hands on CNN and finally use CNN to achieve significant accuracy using Multi-Class Sentiment Analysis. We have also used Glove vectors for getting the word2vec representation of the datasets.

The project aims at making a tradeoff between accuracy and training time because the pre-developed models for emotion detection and recognition via text are slow to train. And, since this is relatively new field compared to emotion recognition via facial expressions and speech recognition, there is scope for the new model developed during the tenure of this project to improve accuracy using some additional new technique if discovered. So, using the customised learning rate with an initial large value and having the value downscaled whenever the model stopped learning rate. Besides, using FastText helped reduce the training even further. This experimental technique helped achieve comparable accuracy in almost one-fifth of the training time. After getting good results with binary datasets, emotions were expanded 7 emotions.

This project aims at developing an algorithm which can give results comparable to the state-of-the-art accuracy with a significant drop in training time. The main focus of the project was to develop and implement various algorithms to detect and recognise emotion from the text. The system so developed can be used as an alternative in case of emergency.

PROBLEM STATEMENT :

The problem statement is to develop a deep learning model which can detect and recognise multiple emotions via text. The model must be reliable with adequate accuracy and should be efficient enough so that a real-time implementation can be obtained. The model should be able to accurately recognise the given words for which it is trained on any individual.

CONTRIBUTIONS :

The objective of the thesis is to lay hands on various models deep learning models for emotion detection and recognition via text and see which complements with the adaptive learning rate achieving similar accuracy with a significant drop in training time. In developing these models, several other effects have been taken into consideration as how to model the appropriate of the model so that the model is able to extract maximum information with minimum model layers. Besides, working on sarcastic comments and incomplete sentences haven't been done yet. That's another thing we worked upon. These models will be discussed in the subsequent chapters.

MOTIVATION :

Emotion detection and recognition have been worked upon but a significant amount of the work focused on using human speech and facial expressions as the dataset. We decided to use text data and achieving state-of-the-art accuracy with a significant drop in training time. Besides, the model was trained on sarcastic comments and incomplete sentences.

LAYOUT OF THE REPORT :

The preprocessing step is explained in chapter 2, feature extraction using CNN and RNN is explained in chapter 3 and finally, the classification techniques are introduced in chapter 4. After this, the development of web application and dataset selection is given in chapter 5 and the methodology and results are explained in chapter 6 and 7 respectively

INTRODUCTION :

The objective of this project is to detect and recognise the emotions by analyzing the text. The emotion recognition so developed can be used for validating and improving its performance of validation and testing in case of incomplete sentences. It also has the potential to recognise emotions from sarcastic comments. Reduced training time is another advantage of the developed model.

Language plays an import role in the field of communication, as through languages you can express your feeling emotions etc. Emotion involves feelings, behaviours, experience and cognition. Emotion could be any strong feelings through some circumstances or mood or relationship.

Exchange of emotion can be done through text, feelings, speech. A human can recognize their feelings, emotions but this is a challenge that how a computer recognize humans feelings in the form of text.

Now here Human-computer interaction plays an important role in the field of digitalization. Every second we have a massive amount of data on the internet now the challenge is how to digitalize this data and how to pick emotions.

Sentiment Analysis is a recent field of research that is closely related to emotion detection and recognition from text. There are two types of analysis available to detect emotion. 1. Sentiment analysis 2. Emotion analysis.

[7] In sentiment analysis we can detect [4] positive, negative or neutral feelings from the given user input text. Emotion could be one word or a bunch of words, it has no specific hierarchy to express emotion. In emotion analysis, we can detect types of feelings such as happy, sad, anger, disgust, fear and surprise(remove extra emotions) from the given user input text. It has some other emotions which fall in the secondary and territory category.

We will work on the sentence level to find emotions from the text. While reading a book or sentence you can find the writer feeling through text. If “A person is happy” you can use positive feel positive response. If “the person is not happy” there would be different concept either the person is frustrated, sad or angry.

So from the text, you can understand the user feel. There are different applications like if an employee sends a harsh email to a colleague or another employee when the employee read the email he will understand the email and through this way, he will protect his job.

Emotion detection can also help the marketer to help strategies for customer to build good relationship management, product service and product delivery. Psychologists can also get benefit from being able to infer people’s emotions based on the text that they write which they can use to predict their state of mind.

Recognition is a big challenge for human as well as for the computer. Here sometimes you cannot recognize a person through their own emotions. While the machine needs some advanced algorithm to recognize data to detect data.

There are two types of recognition method which is hard Sensing and soft sensing. In hard sensing, the sensor can detect emotion like brain signal, heart rate, eye gaze etc, while in soft sensing sensor can detect emotion from software like email, text messaging, desktop, social interaction etc.
In this report, we will pick out text as text emotions because the communication is between human and machine. In a survey report, 68% of people use text messaging to family or friends while only 49% of people use face to face communication.

From face to face talking you will find person impression directly while in text messaging this is not such easy to find the person impressions, so the computer needs such kind of algorithm which can detect emotions.

Here split a sentence on the basis of Noun, pronoun, verb, adverb and some phrases and then find the emotions. A sentence like “Hooray we won the match.” Now there is a relationship between noun ”Match” and verb “won”..

In this approach, first analyzing the given input text and then detect emotion’s type of that particularly given text. Detecting emotion’s type from the text is quite challenging because it can’t give 100% perfect result.

Face expression gives better result compare to the textual expression. For the human-computer interaction (HCI), detect and recognize the emotion of the given input text plays a key role. [1] Emotions may be articulated by a human’s speech, facial expression and written text known as speech, facial and text-based emotion respectively.

Most of the work has been done by speech and facial emotion detection and recognition but text-based emotion detection system is also very attractive for research side because nowadays most of the people write their own expression on their person’s blogs, product reviews/journals, comments etc.

[7] Emotion detection and analysis has been widely researched in psychology, finding happy planet index of the particular country and emotion marketing. By using Emotion Detection, we can generate Happy Planet Index according to the level of Happiness and Healthy lifestyle of the people among the other countries, and also give the rank of the countries according to their Happy Planet Index.

Emotion Detection also used in Emotion Marketing. By using Emotional Marketing we can identify which type of people buys which type of products. Emotionally rich text can be found on product reviews, personal blogs/journals, social network websites etc.

Recognizing textual [2] emotions is a major challenge for both humans and machines because people may not be able to recognize or state their own emotions at certain times or machines need to have a proper position for emotion modelling and also need advanced natural language processing for developing the emotion models. Here in this **thesis project**, we use natural language processing based on knowledge approach to identify the emotion type of that particular text.

The further report is organised as follows. We discuss related works in chapter 2 and in chapter 3, we briefly describe the necessary background work required to understand the concepts involved with the methods. In chapter 4, we discuss the approach and the algorithms. Chapter 5 explains the implementation details such as hyperparameter details. Chapter 6 shows the results of our experiments followed by future works and conclusion in chapter 7 and 8 respectively

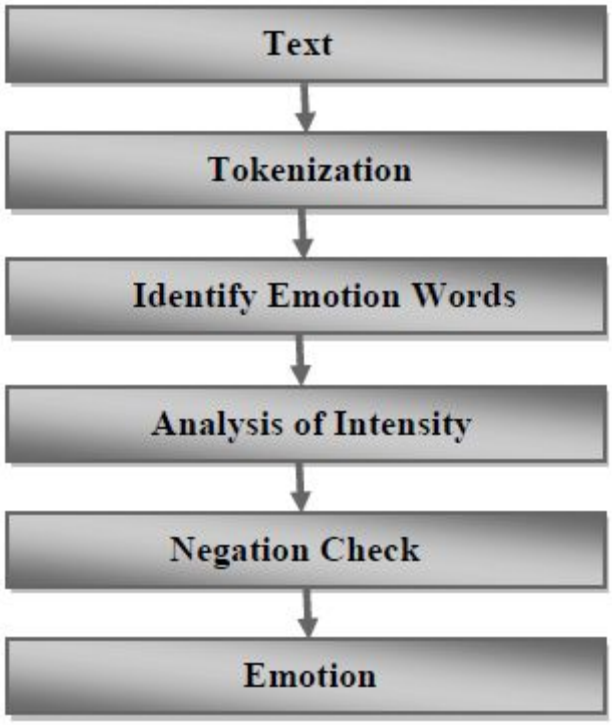
RELATED WORK :

Emotion classification and detection are closely related to Sentimental analysis. Sentimental deals with identifying the positive, negative and neutral nature of the text whereas the Emotion Analysis deals with six basic Ekman emotions. Emotion Detection can be classified as [2] Keyword based Detection and Learning based Detection.

A. Keyword-based Detection :

In this approach, emotion detection is done by extracting emotional keywords from the text. These keywords are matched with the knowledge base or the dictionary such as Thesaurus to find the emotional expression. Thesaurus is a reference work which contains synonyms, antonyms and subset of sunsets.

However, Keyword based Detection has drawback such as (1) unable to identify emotion from the sentence that does not contain any emotional keyword, (2) ambiguity with the the the definition of the keyword as it differs from its usage and context and (3) lack of [2] linguistic information.



[1] Figure 1. Keyword Spotting Technique

B. Learning based Detection :

In this approach, the system is trained to identify the emotion. Deep Learning allows the system to understand the semantic and structure of sentence also the interdependency of the sentence. Emotion dataset is initially constructed which is tagged. This tagged dataset is then fed to the neural network which trains the dataset for more accuracy and handles new data.

There are various options for choosing the training model such as [6] Convolution Neural Network and Recurrent Neural Network. After training the neural network analytic reports are generated until the desired accuracy is not achieved.

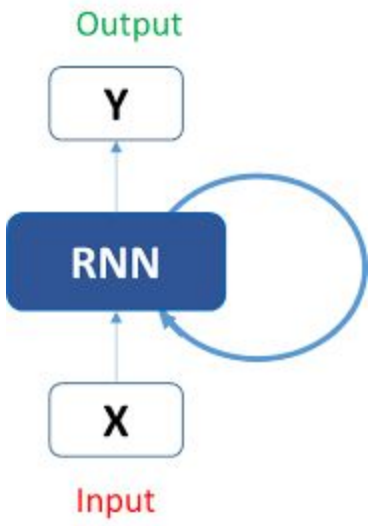


Figure 2. Simple Recurrent Neural Network

DATASET :

IMDB Dataset :

This is a dataset for binary sentiment classification containing substantially more data than previous benchmark datasets. We provide a set of 25,000 highly polar movie reviews for training and 25,000 for testing. There is additional unlabeled data for use as well. Raw text and an already processed bag of words formats are provided. See the README file contained in the release for more details.

APPLICATIONS :

Sentiment analysis also called opinion mining, it focuses on the retrieval of information from the text. Now the computer should be able to detect useful information from text. It has advantages

like to collect useful information from blogs, website and through this information decision taken that either the customer is happy or not. Sentiment analysis is useful for online shopping where to find useful reviews of customer and through these reviews to decide whether the customer is happy or not.

Document consist of many sentences and the reader read through different ways like some time increase the pitch, volume and sometimes different languages people pronounce in different ways. The text to speech generation is a new era application which we enter to the modern world. And through this application, you convert your text to exact speech to pronounce that word.

Mood detection through text has a high advantage in the modern world. Through this way you can increase the knowledge of robot and find the emotion of robots, it also works in automatic answering, and in a dialogue system. Sometime when sending an email the email server replies you automatically.

If a person wants to purchase a product from a company and the person just go the website and instead of reading all the customer reviews and wasting the time of reading the customer just to understand the product usage and usefulness through only one emotion.

It is difficult to find emotions in a sentence rather than the case where there is already an emotional word. It is easy to find emotions in a text document, paragraph etc there may contain many emotions. If the recognition unit is small like keyword it is easy to find emotion but in a sentence, it may have one intention at a time. We also found such kind of emotions detection system in robots, chatting system etc.

There are a lot of research works in the field of Emotion detection through text; Different people use the different sentence to express their feelings like “Oh my God you make this program” show some surprising emotions.

Emotion can be detected through formula let say, E set of possible events, A set of authors, T representation of emotion in text and r be the function to reflect emotion e from the author it is formulated as i.e. r: A * T→ E.

Maybe it could be the best but the problem is new emotion is added day by day and there is no specific standard for emotion like recently in 2015 it says there are only 4 emotions. Here we classify emotions into two categories coarse-grained and fine-grained.

Hancock in (2007) used linguistic inquiry and word count (LIWC) analysis and classifies emotions into positive and negative. Negative emotion expressed in more effective words and positive expression expressed through exclamation mark. But the disadvantage of this method is, that is limited for only happy and sad emotions. Different persons show different states of emotion and classify emotion.

FUTURE WORK :

Emotion Detection is the most [1] important field of research in human-computer interaction. Enough amount of work has been done by researchers to detect emotion from facial and audio information whereas recognizing emotions from textual data is still a fresh and hot research area.

[1] In this paper, methods which are being used to detect emotion from the text are reviewed along with their limitations and a new system architecture is proposed, which would perform efficiently.

CONCLUSION:

Emotion detection is an important part of the field of Human-Computer Interaction. Due to the high usage of social networks or online shopping, many researchers want to work in this field. To classify emotion we need some quick algorithm to detect emotion instead of reading all the websites or blogs.

BIBLIOGRAPHY :

[1] Shaheen, Shadi, Wassim El-Hajj, Hazem Hajj, and Shady Elbassuoni. "Emotion recognition from text based on automatically generated rules." In Data Mining Workshop (ICDMW), 2014 IEEE International Conference on, pp. 383-392. IEEE, 2014.

[2] Tilakraj, Manasa M., Deepika D. Shetty, M. Nagarathna,K. Shruthi, and Sougandhika Narayan. "Emotion Finder:Detecting Emotions From Text, Tweets and Audio."

[3] Povoda, Lukas, Akshaj Arora, Sahitya Singh, Radim Burget, and Malay Kishore Dutta. "Emotion recognition from helpdesk messages." In Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2015 7th International Congress on, pp. 310-313. IEEE, 2015.

[4] Jain, Vinay Kumar, Shishir Kumar, Neha Jain, and Payal Verma. "A Novel Approach to Track Public Emotions Related to Epidemics In Multilingual Data." In 2nd International Conference and Youth School Information Technology and Nanotechnology (ITNT 2016), Russia, pp. 883-889. 2016.

[5] Weerasooriya, Tharindu, Nandula Perera, and S. R. Liyanage. "A method to extract essential keywords from a tweet using NLP tools." In Advances in ICT for Emerging Regions (ICTer), 2016 Sixteenth International Conference on, pp. 29-34. IEEE, 2016.

[6] Shivhare, Shiv Naresh, and Saritha Khethawat. "Emotion detection from text." arXiv preprint arXiv:1205.4944 (2012).

[7] Prof. Hardik S. Jayswal, Dhruvi D. Gosai and Himangini J. Gohil. “A REVIEW ON A EMOTION DETECTION AND RECOGNIZATION FROM TEXT USING NATURAL LANGUAGE PROCESSING”. In Research Gate, April 2018 Conference.

[8] Kashif khan, Sher Hayat and Muhammad Ejaz khan. “Emotion Detection through Text: Survey”. The International Journal Research Publication’s, Research Journal of Science and Management.

[9] https://monkeylearn.com/sentiment-analysis/

[10] https://blog.algorithmia.com/introduction-sentiment-analysis/

[11] https://medium.com/seek-blog/your-guide-to-sentiment-analysis-344d43d225a7

[12] http://anie.me/On-Torchtext/

[13] http://mlexplained.com/2018/02/08/a-comprehensive-tutorial-to-torchtext/
[14] https://github.com/spro/practical-pytorch

[15] https://github.com/Shawn1993/cnn-text-classification-pytorch

[16] https://spacy.io/

[17] https://monkeylearn.com/blog/word-embeddings-transform-text-numbers/

[18] https://nlp.stanford.edu/projects/glove/

[19]

[20]

[21]

[22]

[23]

[24]

[25]

[26]

Simple Sentiment Analysis :(Module I)

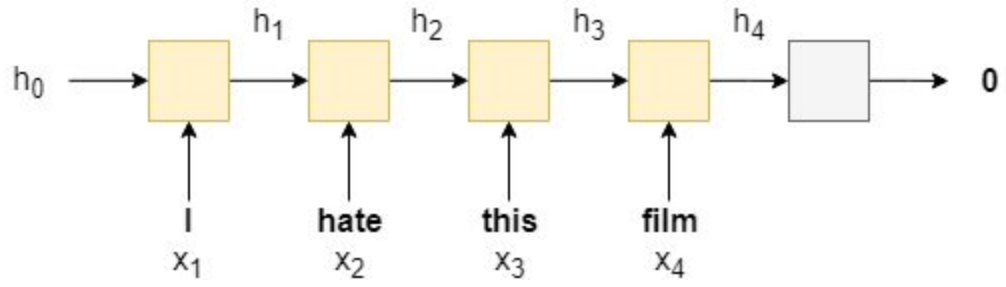
RNN :

We'll be using a **recurrent neural network** (RNN) as they are commonly used in analysing sequences. An RNN takes in a sequence of words, $X = \{ x_1, .., x_t \}$, one at a time, and produces a *hidden state*, h, for each word. We use the RNN *recurrently* by feeding in the current word x_t as well as the hidden state from the previous word, h_{t-1}, to produce the next hidden state, h_t.

$ht = RNN(x_t , h_{t-1})$

Once we have our final hidden state, h_T, (from feeding in the last word in the sequence, x_T) we feed it through a linear layer, f, (also known as a fully connected layer), to receive our predicted sentiment, $y = f(h_T)$.

Below shows an example sentence, with the RNN predicting zero, which indicates negative sentiment. The RNN is shown in orange and the linear layer shown in silver. Note that we use the same RNN for every word, i.e. it has the same parameters. The initial hidden state, h_0, is a tensor initialized to all zeros.



Preparing Data :

One of the main concepts of TorchText is the Field. These define how your data should be processed. In our sentiment classification task, the data consists of both the raw string of the review and the sentiment, either "pos" or "neg". The parameters of a Field specify how the data should be

processed. We use the TEXT field to define how the review should be processed, and the LABEL field to process the sentiment.

Our TEXT field has tokenize='spacy' as an argument. This defines that the "tokenization" (the act of splitting the string into discrete "tokens") should be done using the spaCy[15] tokenizer. If no tokenize argument is passed, the default is simply splitting the string into spaces. LABEL is defined by a LabelField, a special subset of the Field class specifically used for handling labels. We will explain the datatype argument later. We also set the random seeds for reproducibility.

Another handy feature of TorchText is that it has support for common datasets used in natural language processing (NLP). The code automatically downloads the IMDb dataset and splits it into the canonical train/test splits as torchtext.datasets objects. It processes the data using the Fields we have previously defined. The IMDb dataset consists of 50,000 movie reviews, each marked as being a positive or negative review.

The IMDb dataset only has train/test splits, so we need to create a validation set. We can do this with the .split() method. By default this splits 70/30, however by passing a split_ratio argument, we can change the ratio of the split, i.e. a split_ratio of 0.8 would mean 80% of the examples make up the training set and 20% make up the validation set. We also pass our random seed to the random_state argument, ensuring that we get the same train/validation split each time.

Next, we have to build a *vocabulary*. This is an effectively a look-up a table where every unique word in your data set has a corresponding *index* (an integer).

We do this as our machine learning model cannot operate on strings, only numbers. Each *index* is used to construct a *one-hot* vector for each word. A one-hot vector is a vector where all of the elements are 0, except one, which is 1, and dimensionality is the total number of unique words in your vocabulary, commonly denoted by V.

word	index	one-hot vector
I	0	[1, 0, 0, 0]
hate	1	[0, 1, 0, 0]
this	2	[0, 0, 1, 0]
film	3	[0, 0, 0, 1]

The number of unique words in our training set is over 100,000, which means that our one-hot vectors will have over 100,000 dimensions! This will make training slow and possibly won't fit onto your GPU (if you're using one).

There are two ways to effectively cut down our vocabulary, we can either only take the top n most common words or ignore words that appear less than m times. We'll do the former, only keeping the top 25,000 words.

What do we do with words that appear in examples but we have cut from the vocabulary? We replace them with a special unknown or <unk> token. For example, if the sentence was "This film is great and I love it" but the word "love" was not in the vocabulary, it would become "This film is great and I <unk> it". The code builds the vocabulary, only keeping the most common max_size tokens.

When testing any machine learning system you do not want to look at the test set in any way. We do not include the validation set as we want it to reflect the test set as much as possible.

The vocab size 25002 and not 25000. One of the additional tokens is the <unk> token and the other is a <pad> token.

When we feed sentences into our model, we feed a *batch* of them at a time, i.e. more than one at a time, and all sentences in the batch need to be the same size. Thus, to ensure each sentence in the batch is the same size, any shorter than the longest within the batch are padded.

sent1	sent2
I	This
hate	film
this	sucks
film	<pad>

We can also view the most common words in the vocabulary and their frequencies.

The final step of preparing the data is creating the iterators. We iterate over these in the training/evaluation loop, and they return a batch of examples (indexed and converted into tensors) at each iteration.

We'll use a BucketIterator which is a special type of iterator that will return a batch of examples where each example is of a similar length, minimizing the amount of padding per example.

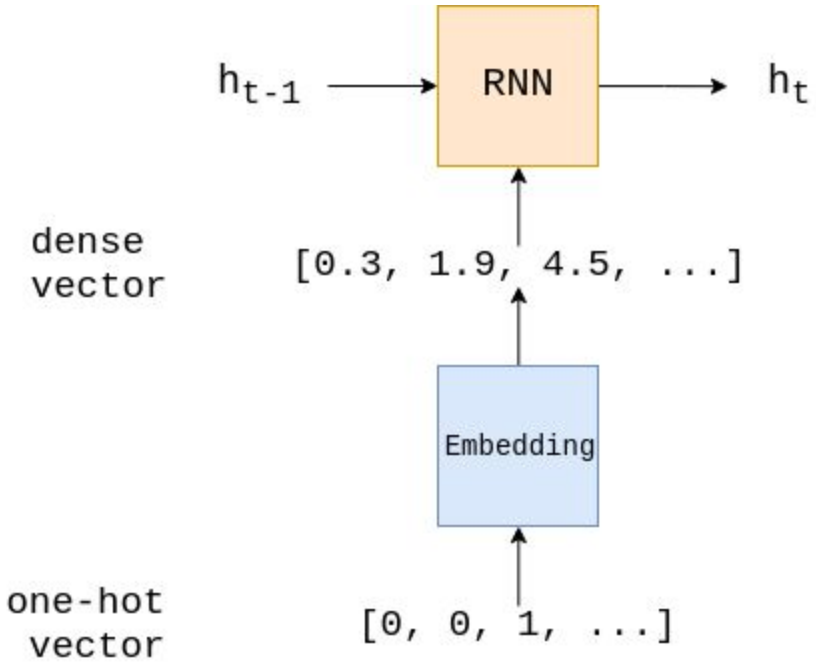
We also want to place the tensors returned by the iterator on the GPU (if you're using one). PyTorch handles this using torch. a device, we then pass this device to the iterator.

Build the Model :

The next stage is building the model that we'll eventually train and evaluate. There is a small amount of boilerplate code when creating models in PyTorch, note how our RNN class is a sub-class of nn.Module and the use of super. Within the __init__ we define the *layers* of the module. Our three layers are an *embedding* layer, our RNN, and a *linear* layer. All layers have their parameters initialized to random values unless explicitly specified.

The embedding layer is used to transform our sparse one-hot vector (sparse as most of the elements are 0) into a dense embedding vector (dense as the dimensionality is a lot smaller and all the elements are real numbers). This embedding layer is simply a single fully connected layer. As well as reducing the dimensionality of the input to the RNN, there is the theory that words which have a similar impact on the sentiment of the review are mapped close together in this dense vector space. For more information about word embeddings, see here[16].

The RNN layer is our RNN which takes in our dense vector and the previously hidden state h_{t-1}, which it uses to calculate the next hidden state, h_t.



Finally, the linear layer takes the final hidden state and feeds it through a fully connected layer, $f(h_t)$, transforming it to the correct output dimension. The forward method is called when we feed examples into our model. Each batch, text, is a tensor of size ***[sentence length, batch size]***. That is a batch of sentences, each having each word converted into a one-hot vector.

You may notice that this tensor should have another dimension due to the one-hot vectors, however, PyTorch conveniently stores a one-hot vector as it's index value, i.e. the tensor representing a sentence is just a tensor of the indexes for each token in that sentence. The act of converting a list of tokens into a list of indexes is commonly called *numericalizing*.

The input batch is then passed through the embedding layer to get embedded, which gives us a dense vector representation of our sentences. embedded is a tensor of size ***[sentence length, batch size, embedding dim]*** embedded is then fed into the RNN. In some frameworks you must feed the initial hidden state, h_0, into the RNN, however in PyTorch, if no initial hidden state is passed as an argument it defaults to a tensor of all zeros.

The RNN returns 2 tensors, output of size ***[sentence length, batch size, hidden dim]*** and hidden of size ***[1, batch size, hidden dim]***. the output is the concatenation of the hidden state from every time step, whereas hidden is simply the final hidden state. We verify this using the assert statement. Note the squeeze method, which is used to remove a dimension of size 1.

Finally, we feed the last hidden state, hidden, through the linear layer, fc, to produce a prediction.

We then create an instance of our RNN class. The input dimension is the dimension of the one-hot vectors, which is equal to the vocabulary size. The embedding dimension is the size of the dense word vectors. This is usually around 50-250 dimensions but depends on the size of the vocabulary.

The hidden dimension is the size of the hidden states. This is usually around 100-500 dimensions but also depends on factors such as on the vocabulary size, the size of the dense vectors and the complexity of the task. The output dimension is usually the number of classes, however, in the case of only 2 classes, the output value is between 0 and 1 and thus can be 1-dimensional, i.e. a single scalar real number. We create a function that will tell us how many trainable parameters our model has so we can compare the number of parameters across different models.

TRAIN THE MODEL :

First, we'll create an optimizer. This is the algorithm we use to update the parameters of the module. Here, we'll use *stochastic gradient descent* (SGD). The first argument is the parameters will be updated by the optimizer, the second is the learning rate, i.e. how much we'll change the parameters by when we do a parameter update.

First, we'll create an optimizer. This is the algorithm we use to update the parameters of the module. Here, we'll use *stochastic gradient descent* (SGD). The first argument is the parameters will be updated by the optimizer, the second is the learning rate, i.e. how much we'll change the parameters by when we do a parameter update.

Our criterion function calculates the loss, however, we have to write our function to calculate the accuracy. This function first feeds the predictions through a sigmoid layer, squashing the values between 0 and 1, we then round them to the nearest integer. This rounds any value greater than 0.5 to 1 (a positive sentiment) and the rest to 0 (a negative sentiment). We then calculate how many rounded predictions equal the actual labels and average it across the batch.

The train function iterates over all examples, one batch at a time. model.train() is used to put the model in "training mode", which turns on *dropout* and *batch normalization*. Although we aren't using them in this model, it's good practice to include it. For each batch, we first zero the gradients. Each

parameter in a model has a grad attribute which stores the gradient calculated by the criterion. PyTorch does not automatically remove (or "zero") the gradients calculated from the last gradient calculation, so they must be manually zeroed.

We then feed the batch of sentences, batch.text, into the model. Note, you do not need to do model.forward(batch.text), simply calling the model works. The squeeze is needed as the predictions are initially size ***[batch size, 1]***, and we need to remove the dimension of size 1 as PyTorch expects to our criterion function to be of size ***[batch size]***. The loss and accuracy are then calculated using our predictions and the labels, batch.label, with the loss being averaged over all examples in the batch.

We calculate the gradient of each parameter with loss.backward(), and then update the parameters using the gradients and optimizer algorithm with optimizer.step(). The loss and accuracy are accumulated across the epoch, the .item() method is used to extract a scalar from a tensor which only contains a single value. Finally, we return the loss and accuracy, averaged across the epoch. The length of an iterator is the number of batches in the iterator.

We set dtype=torch.float because TorchText sets tensors to be LongTensors by default, however, our criterion expects both inputs to be FloatTensors. Setting the dtype to be torch.float, did this for us. The alternative method of doing this would be to do the conversion inside the train function by passing batch.label.float() instead of batch.label to the criterion.

Evaluate is similar to a train, with a few modifications as you don't want to update the parameters when evaluating. model.eval() puts the model in "evaluation mode", this turns off *dropout* and *batch normalization*. Again, we are not using them in this model, but it is good practice to include them. No gradients are calculated on PyTorch operations inside the with no_grad() block. This causes less memory to be used and speeds up computation. The rest of the function is the same as train, with the removal of optimizer.zero_grad(), loss.backward() and optimizer.step(), as we do not update the model's parameters when evaluating.

We then train the model through multiple epochs, an epoch being a complete pass through all examples in the training and validation sets. At each epoch, if the validation loss is the best we have seen so far, we'll save the parameters of the model and then after training has finished we'll use that model on the test set.

```
Epoch: 01 | Epoch Time: 0m 18s
        Train Loss: 0.694 | Train Acc: 50.12%
         Val. Loss: 0.696 |  Val. Acc: 50.17%
Epoch: 02 | Epoch Time: 0m 19s
        Train Loss: 0.693 | Train Acc: 49.73%
         Val. Loss: 0.696 |  Val. Acc: 51.01%
Epoch: 03 | Epoch Time: 0m 17s
        Train Loss: 0.693 | Train Acc: 50.22%
         Val. Loss: 0.696 |  Val. Acc: 50.87%
```

Epoch: 04 | Epoch Time: 0m 18s
	Train Loss: 0.693 | Train Acc: 49.94%
	 Val. Loss: 0.696 | Val. Acc: 49.91%
Epoch: 05 | Epoch Time: 0m 18s
	Train Loss: 0.693 | Train Acc: 50.07%
	 Val. Loss: 0.696 | Val. Acc: 51.00%

Test Loss: 0.708 | Test Acc: 47.87%

ADVANCED SENTIMENT ANALYSIS :

PREPARING THE DATA :

We'll be using *packed padded sequences*, which will make our RNN only process the non-padded elements of our sequence, and for any padded element, the output will be a zero tensor. To use packed padded sequences, we have to tell the RNN how long the actual sequences are. We do this by setting include_lengths = True for our TEXT field. This will cause batch.text to now be a tuple with the first element being our sentence (a numericalized tensor that has been padded) and the second element being the actual lengths of our sentences.

Next is the use of pre-trained word embeddings. Now, instead of having our word embeddings initialized randomly, they are initialized with these pre-trained vectors. We get these vectors simply by specifying which vectors we want and passing it as an argument to build_vocab. TorchText handles downloading the vectors and associating them with the correct words in our vocabulary. Here, we'll be using the "glove.6B.100d" vectors". the glove is the algorithm used to calculate the vectors, go here[17] for more. 6B indicates these vectors were trained on 6 billion tokens and 100d indicates these vectors are 100-dimensional.

The theory is that these pre-trained vectors already have words with a 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. By default, TorchText will initialize words in your vocabulary but not in your pre-trained embeddings to zero. We don't want this and instead, initialize them randomly by setting unk_init to torch.Tensor.normal_. This will now initialize those words via a Gaussian distribution.

As before, we create the iterators, placing the tensors on the GPU if one is available. Another thing for packed padded sequences all of the tensors within a batch needs to be sorted by their lengths. This is handled in the iterator by setting sort_within_batch = True.

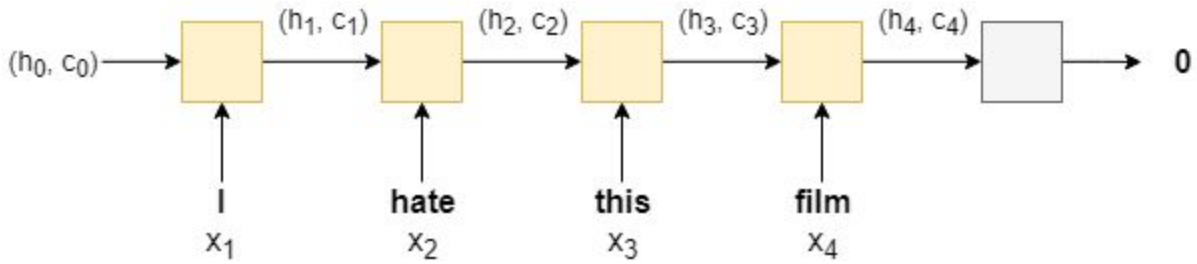
BUILD THE MODEL

The model features the most drastic changes.

DIFFERENT RNN ARCHITECTURE :

We'll be using a different RNN architecture called a Long Short-Term Memory (LSTM). Why is an LSTM better than a standard RNN? Standard RNNs suffer from the vanishing gradient problem[19]. 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 multiple *gates* which control the flow of information into and out of the memory. For more information, go here[20]. 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 (with the embedding layers omitted):



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. $y = f(h_T)$.

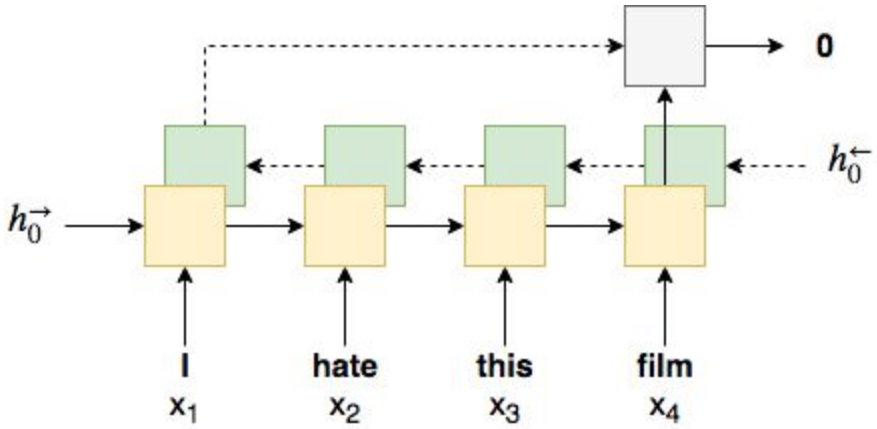
BIDIRECTIONAL RNN :

The concept behind a bidirectional RNN is simple. As well as 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). At time step t, the forward RNN is processing word x_t, and the backward RNN is processing word x_{T-t+1}.

In PyTorch, the hidden state (and cell state) tensors returned by the forward and backward RNNs are stacked on top of each other in a single tensor.

We make our sentiment prediction using a concatenation of the last hidden state from the forward RNN (obtained from final word of the sentence), h_T, and the last hidden state from the backward RNN (obtained from the first word of the sentence), h_T, i.e. $y = f(h_T, h_T')$

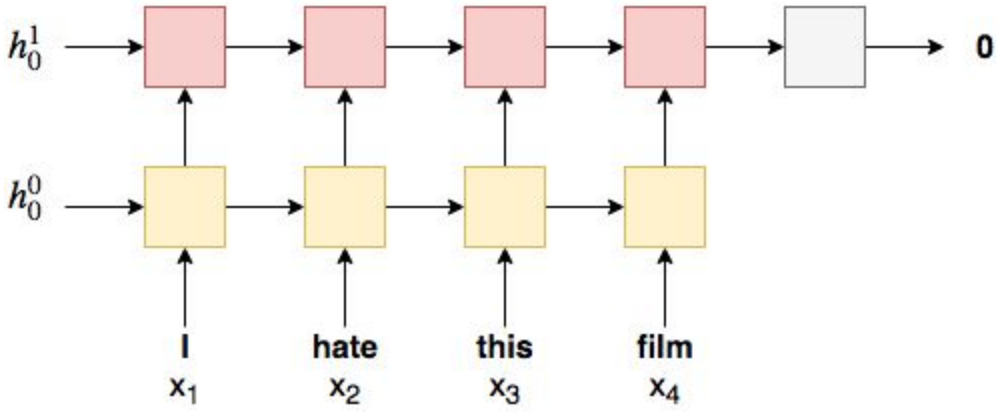
The image below shows a bidirectional RNN, with the forward RNN in orange, the backward RNN in green and the linear layer in silver.



MULTI-LAYER RNN :

Multi-layer RNN (also called *deep RNN*) is 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 its own initial hidden state, h^L_0.



REGULARIZATION:

Although we've added improvements to our model, each one adds additional parameters. Without going into overfitting into too much detail, the more parameters you have in your model, the higher the probability that your model will overfit (memorize the training data, causing a low training error but high validation/testing error, i.e. poor generalization to new, unseen examples). To combat this, we use regularization. More specifically, we use a method of regularization called *dropout*. Dropout works by randomly *dropping out*(setting to 0) neurons in a layer during a forward pass. The probability that each neuron is dropped out is set by a hyperparameter and each neuron with dropout applied is considered independent. One theory about why dropout works is that a model

with parameters dropped out can be seen as a "weaker" (fewer parameters) model. The predictions from all these "weaker" models (one for each forward pass) get averaged together within the parameters of the model. Thus, your one model can be thought of as an ensemble of weaker models, none of which are over-parameterized and thus should not overfit.

IMPLEMENTATION DETAILS :

Another addition to this model is that we are not going to learn the embedding for the <pad> token. This is because we want to explicitly tell our model that padding tokens are irrelevant to determining the sentiment of a sentence. This means the embedding for the pad token will remain at what it is initialized to (we initialize it to all zeros later). We do this by passing the index of our pad token as the padding_idx argument to the nn.Embedding layer.

To use an LSTM instead of the standard RNN, we use nn.LSTM instead of nn.RNN. Also, note that the LSTM returns the output and a tuple of the final hidden state and the final cell state, whereas the standard RNN only returned the output and final hidden state.

As the final hidden state of our LSTM has both a forward and a backward component, which will be concatenated together, the size of the input to the nn.linear layer is twice that of the hidden dimension size. Implementing bidirectionality and adding additional layers are done by passing values for the num_layers and bidirectional arguments for the RNN/LSTM.

Dropout is implemented by initializing an nn.Dropout layer (the argument is the probability of dropping out each neuron) and using it within the forward method after each layer we want to apply dropout to. **Note**: never use dropout on the input or output layers (textor fc in this case), you only ever want to use dropout on intermediate layers. The LSTM has a dropout argument which adds dropout on the connections between hidden states in one layer to hidden states in the next layer. As we are passing the lengths of our sentences to be able to use packed padded sequences, we have to add a second argument, text_lengths, to forward.

Before we pass our embeddings to the RNN, we need to pack them, which we do with nn.utils.rnn.packed_padded_sequence. This will cause our RNN to only process the non-padded elements of our sequence. The RNN will then return packed_output (a packed sequence) as well as the hidden and cell states (both of which are tensors). Without packed padded sequences, hidden and cell is tensors from the last element in the sequence, which will most probably be a pad token, however, when using packed padded sequences they are both from the last non-padded element in the sequence.

We then unpack the output sequence, with nn.utils.rnn.pad_packed_sequence, to transform it from a packed sequence to a tensor. The elements of output from padding tokens will be zero tensors (tensors where every element is zero). Usually, we only have to unpack output if we are going to use it later on in the model. Although we aren't in this case, we still unpack the sequence just to show how it is done.
The final hidden state, hidden, has a shape of ***[num layers * num directions, batch size, hid dim]***. These are ordered: **[forward_layer_0, backward_layer_0, forward_layer_1, backward_layer 1, ..., forward_layer_n, backward_layer n]**. As we want the final (top) layer forward and backward

hidden states, we get the top two hidden layers from the first dimension, hidden[-2,:,:]and hidden[-1,:,:], and concatenate them together before passing them to the linear layer (after applying dropout).

Like before, we'll create an instance of our RNN class, with the new parameters and arguments for the number of layers, bidirectionality and dropout probability. To ensure the pre-trained vectors can be loaded into the model, the EMBEDDING_DIM must be equal to that of the pre-trained GloVe vectors loaded earlier. We get our pad token index from the vocabulary, getting the actual string representing the pad token from the field's pad_tokenattribute, which is <pad> by default.

The model has 4,810,857 trainable parameters. Notice how we have almost twice as many parameters as before! The final addition is copying the pre-trained word embeddings we loaded earlier into the embedding layer of our model. We retrieve the embeddings from the field's vocab and check they're the correct size, ***[vocab size, embedding dim]*** We then replace the initial weights of the embedding layer with the pre-trained embeddings.

```
tensor([[-0.1117, -0.4966,  0.1631,  ...,  1.2647, -0.2753, -0.1325],
        [-0.8555, -0.7208,  1.3755,  ...,  0.0825, -1.1314,  0.3997],
        [-0.0382, -0.2449,  0.7281,  ..., -0.1459,  0.8278,  0.2706],
        ...,
        [-0.0614, -0.0516, -0.6159,  ..., -0.0354,  0.0379, -0.1809],
        [ 0.1885, -0.1690,  0.1530,  ..., -0.2077,  0.5473, -0.4517],
        [-0.1182, -0.4701, -0.0600,  ...,  0.7991, -0.0194,  0.4785]])
```

As our <unk> and <pad> token aren't in the pre-trained vocabulary they have been initialized using unk_init (an $N(0, 1)$ distribution) when building our vocab. It is preferable to initialize them both to all zeros to explicitly tell our model that, initially, they are irrelevant for determining sentiment. We do this by manually setting their row in the embedding weights matrix to zeros. We get their row by finding the index of the tokens, which we have already done for the padding index

```
tensor([[ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
        [-0.0382, -0.2449,  0.7281,  ..., -0.1459,  0.8278,  0.2706],
        ...,
        [-0.0614, -0.0516, -0.6159,  ..., -0.0354,  0.0379, -0.1809],
        [ 0.1885, -0.1690,  0.1530,  ..., -0.2077,  0.5473, -0.4517],
        [-0.1182, -0.4701, -0.0600,  ...,  0.7991, -0.0194,  0.4785]])
```

We can now see the first two rows of the embedding weights matrix have been set to zeros. As we passed the index of the pad token to the padding_idx of the embedding layer it will remain zeros throughout training, however, the <unk> token embedding will be learned.

TRAIN THE MODELS

The only change we'll make here is changing the optimizer from SGD to Adam. SGD updates all parameters with the same learning rate and choosing this learning rate can be tricky. Adam adapts the learning rate for each parameter, giving parameters that are updated more frequently lower learning rates and parameters that are updated infrequently higher learning rates. More information about Adam (and other optimizers) can be found here[21].

To change SGD to Adam, we simply change optim.SGD to optim.Adam, also note how we do not have to provide an initial learning rate for Adam as PyTorch specifies a sensibile default initial learning rate.

As we have set include_lengths = True, our batch.text is now a tuple with the first element being the numericalized tensor and the second element being the actual lengths of each sequence. We separate these into their own variables, text and text_lengths, before passing them to the model. As we are now using dropout, we must remember to use model.train() to ensure the dropout is "turned on" while training.

USER INPUT :
We can now use our model to predict the sentiment of any sentence we give it. As it has been trained on movie reviews, the sentences provided should also be movie reviews. When using a model for inference it should always be in evaluation mode. If this tutorial is followed step-by-step then it should already be in evaluation mode (from doing evaluate on the test set), however we explicitly set it to avoid any risk.

Epoch: 01 | Epoch Time: 0m 28s
	Train Loss: 0.641 | Train Acc: 62.54%
	 Val. Loss: 0.520 | Val. Acc: 74.92%
Epoch: 02 | Epoch Time: 0m 28s
	Train Loss: 0.559 | Train Acc: 71.77%
	 Val. Loss: 0.735 | Val. Acc: 62.60%
Epoch: 03 | Epoch Time: 0m 27s
	Train Loss: 0.479 | Train Acc: 78.00%
	 Val. Loss: 0.403 | Val. Acc: 82.23%
Epoch: 04 | Epoch Time: 0m 28s
	Train Loss: 0.381 | Train Acc: 83.92%
	 Val. Loss: 0.343 | Val. Acc: 85.85%
Epoch: 05 | Epoch Time: 0m 27s
	Train Loss: 0.319 | Train Acc: 86.83%
	 Val. Loss: 0.360 | Val. Acc: 85.59%

```
Test Loss: 0.342 | Test Acc: 86.02%
```

Our predict_sentiment function does a few things:

- sets the model to evaluation mode
- tokenizes the sentence, i.e. splits it from a raw string into a list of tokens
- indexes the tokens by converting them into their integer representation from our vocabulary
- gets the length of our sequence
- converts the indexes, which are a Python list into a PyTorch tensor
- add a batch dimension by un squeezeing
- converts the length into a tensor
- squashes the output prediction from a real number between 0 and 1 with the sigmoid function
- converts the tensor holding a single value into an integer with the item() method
- We are expecting reviews with a negative sentiment to return a value close to 0 and positive reviews to return a value close to 1.

FASTER SENTIMENT ANALYSIS :
We'll implement a model that gets comparable results whilst training significantly faster and using around half of the parameters. More specifically, we'll be implementing the "FastText" model from the paper Bag of Tricks for Efficient Text Classification[22].

PREPARING THE DATA

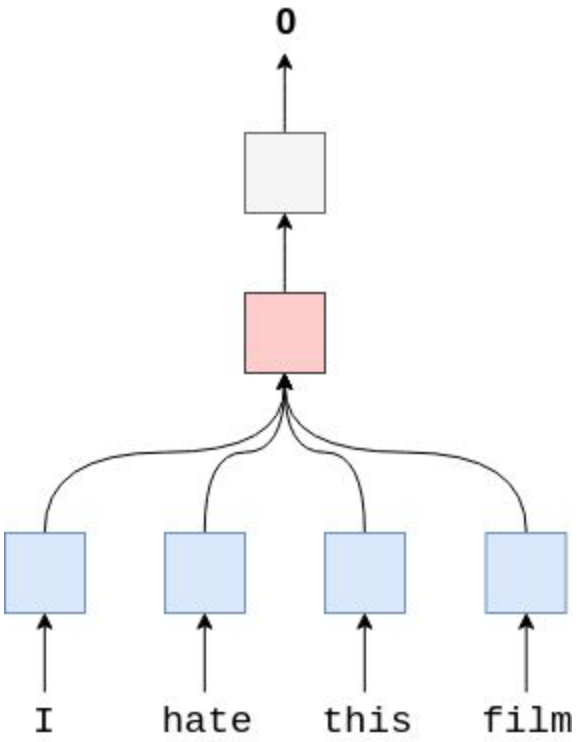
One of the key concepts in the FastText paper is that they calculate the n-grams of an input sentence and append them to the end of a sentence. Here, we'll use bi-grams. Briefly, a bi-gram is a pair of words/tokens that appear consecutively within a sentence. For example, in the sentence "how are you ?", the bi-grams are: "how are", "are you" and "you ?". The generate_bigrams function takes a sentence that has already been tokenized, calculates the bi-grams and appends them to the end of the tokenized list.

TorchText Fields have a preprocessing argument. A function passed here will be applied to a sentence after it has been tokenized (transformed from a string into a list of tokens), but before it has been numericalized (transformed from a list of tokens to a list of indexes). This is where we'll pass our generate_bigrams function. As we aren't using an RNN we can't use packed padded sequences, thus we do not need to set include_lengths = True.

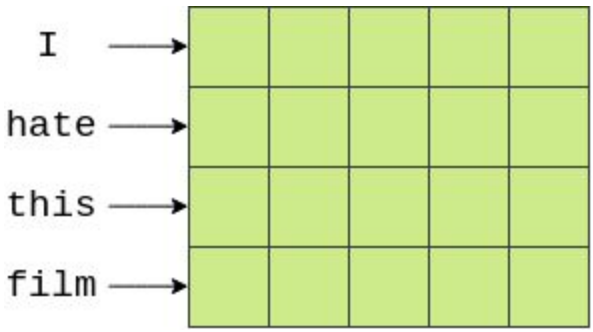
BUILD THE MODEL :

This model has far fewer parameters than the previous model as it only has 2 layers that have any parameters, the embedding layer and the linear layer. There is no RNN component in sight!

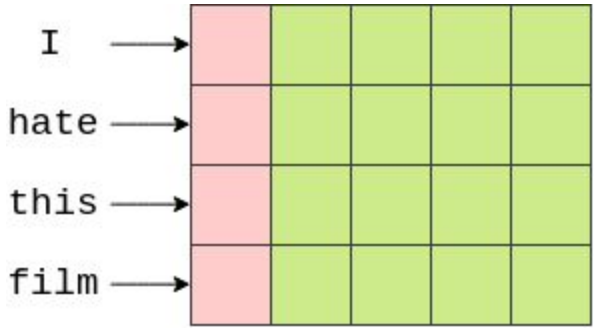
Instead, it first calculates the word embedding for each word using the Embedding layer (blue), then calculates the average of all of the word embeddings (pink) and feeds this through the Linear layer (silver).



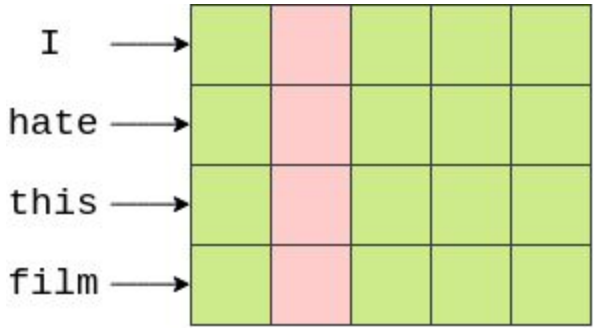
We implement the averaging with the avg_pool2d (average pool 2-dimensions) function. Initially, you may think using a 2-dimensional pooling seems strange, surely our sentences are 1-dimensional, not 2-dimensional? However, you can think of the word embeddings as a 2-dimensional grid, where the words are along one axis and the dimensions of the word embeddings are along with the other. The image below is an example sentence after being converted into 5-dimensional word embeddings, with the words along the vertical axis and the embeddings along the horizontal axis. Each element in this [4x5] tensor is represented by a green block.



The avg_pool2d uses a filter of size embedded.shape[1] (i.e. the length of the sentence) by 1. This is shown in pink in the image below.



We calculate the average value of all elements covered by the filter, then the filter then slides to the right, calculating the average over the next column of embedding values for each word in the sentence.



Each filter position gives us a single value, the average of all covered elements. After the filter has covered all embedding dimensions we get a [1x5] tensor. This tensor is then passed through the linear layer to produce our prediction.

TRAIN THE MODEL :

Epoch: 01 | Epoch Time: 0m 7s
	Train Loss: 0.687 | Train Acc: 57.26%
	 Val. Loss: 0.639 | Val. Acc: 71.30%
Epoch: 02 | Epoch Time: 0m 7s
	Train Loss: 0.652 | Train Acc: 72.59%
	 Val. Loss: 0.516 | Val. Acc: 75.71%
Epoch: 03 | Epoch Time: 0m 7s
	Train Loss: 0.581 | Train Acc: 79.36%
	 Val. Loss: 0.436 | Val. Acc: 79.92%
Epoch: 04 | Epoch Time: 0m 7s
	Train Loss: 0.503 | Train Acc: 83.88%
	 Val. Loss: 0.400 | Val. Acc: 83.41%
Epoch: 05 | Epoch Time: 0m 7s
	Train Loss: 0.437 | Train Acc: 86.86%
	 Val. Loss: 0.392 | Val. Acc: 85.09%

Test Loss: 0.386 | Test Acc: 85.18%

CNN

We will be using a *convolutional neural network* (CNN) to conduct sentiment analysis, implementing the model from Convolutional Neural Networks for Sentence Classification[23].

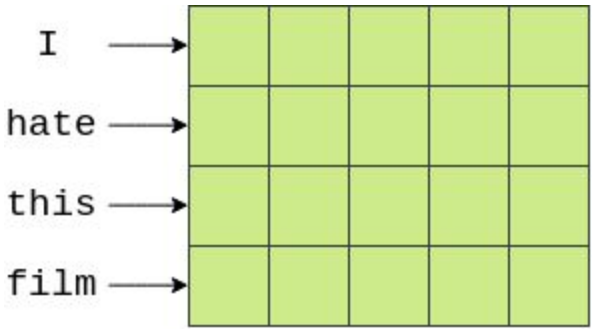
Traditionally, CNNs are used to analyse images and are made up of one or more *convolutional* layers, followed by one or more linear layers. The convolutional layers use filters (also called *kernels* or *receptive fields*) which scan across an image and produce a processed version of the image. This processed version of the image can be fed into another convolutional layer or a linear layer. Each filter has a shape, e.g. a 3x3 filter covers a 3 pixel wide and 3-pixel high area of the image, and each element of the filter has a weight associated with it, the 3x3 filter would have 9 weights. In traditional image processing, these weights were specified by hand by engineers, however, the main advantage of the convolutional layers in neural networks is that these weights are learned via backpropagation.

The intuitive idea behind learning the weights is that your convolutional layers act like *feature extractors*, extracting parts of the image that are most important for your CNN's goal, e.g. if using a CNN to detect faces in an image, the CNN may be looking for features such as the existence of a nose, mouth or a pair of eyes in the image.

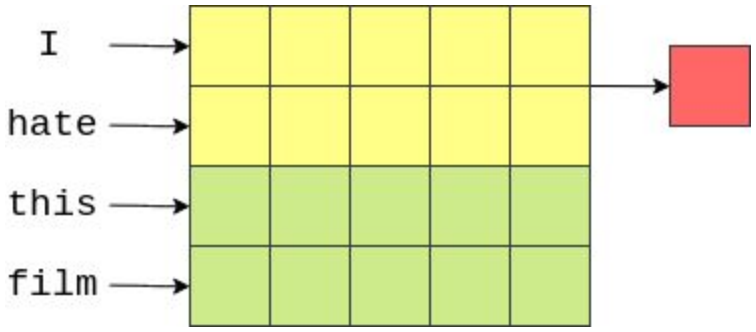
So why use CNNs on text? In the same way that a 3x3 filter can look over a patch of an image, a 1x2 filter can look over 2 sequential words in a piece of text, i.e. a bi-gram. In the previous tutorial we looked at the FastText model which used bi-grams by explicitly adding them to the end of a text, in this CNN model we will instead use multiple filters of different sizes which will look at the bi-grams (a 1x2 filter), tri-grams (a 1x3 filter) and/or n-grams (a 1xn filter) within the text. The intuition here is that the appearance of certain bi-grams, tri-grams and n-grams within the review will be a good indication of the final sentiment.
We will prepare data just like previously. We no longer explicitly need to create the bi-grams and append them to the end of the sentence.

BUILD THE MODEL

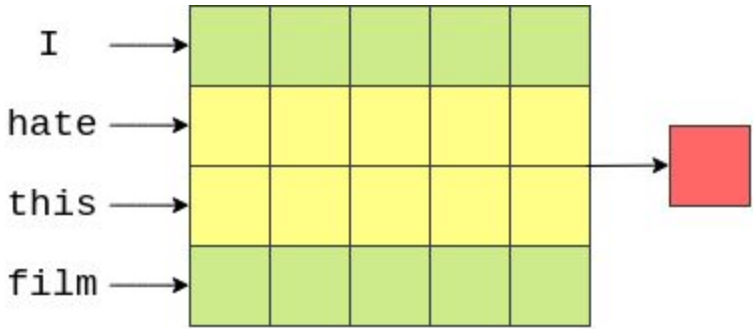
The first major hurdle is visualizing how CNNs are used for text. Images are typically 2 dimensional (we'll ignore the fact that there is a third "colour" dimension for now) whereas text is 1 dimensional. However, we know that the first step in almost all of our previous tutorials (and pretty much all NLP pipelines) is converting the words into word embeddings. This is how we can visualize our words in 2 dimensions, each word along one axis and the elements of vectors across the other dimension. Consider the representation of the embedded sentence below:



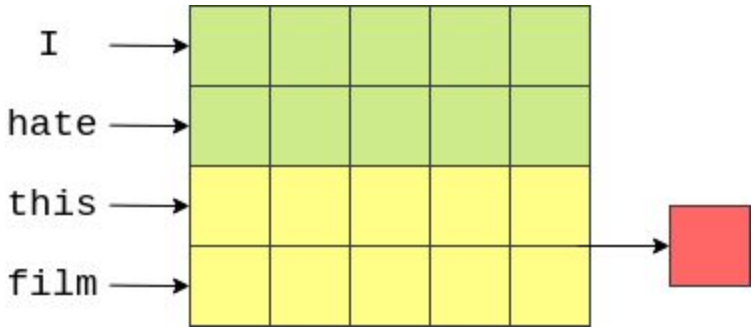
We can then use a filter that is **[n x emb_dim]**. This will cover n sequential words entirely, as their width will be emb_dim dimensions. Consider the image below, with our word vectors, are represented in green. Here we have 4 words with 5-dimensional embeddings, creating a [4x5] "image" tensor. A filter that covers two words at a time (i.e. bi-grams) will be **[2x5]** filter, shown in yellow, and each element of the filter with have a *weight* associated with it. The output of this filter (shown in red) will be a single real number that is the weighted sum of all elements covered by the filter.



The filter then moves "down" the image (or across the sentence) to cover the next bi-gram and another output (weighted sum) is calculated.



Finally, the filter moves down again and the final output for this filter is calculated.

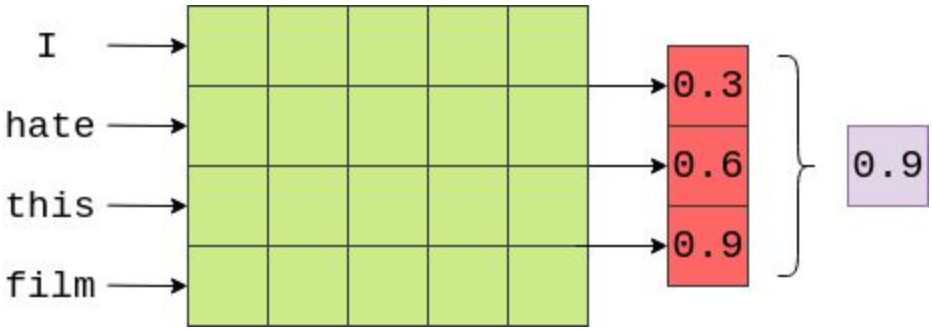


In our case (and in the general case where the width of the filter equals the width of the "image"), our output will be a vector with number of elements equal to the height of the image (or length of the word) minus the height of the filter plus one, 4 - 2 + 1 = 3, in this case.

This example showed how to calculate the output of one filter. Our model (and pretty much all CNNs) will have lots of these filters. The idea is that each filter will learn a different feature to extract. In the above example, we are hoping each of the **[2 x emb_dim]**filters will be looking for the occurrence of different bi-grams.

In our model, we will also have different sizes of filters, heights of 3, 4 and 5, with 100 of each of them. The intuition is that we will be looking for the occurrence of different tri-grams, 4-grams and 5-grams that are relevant for analysing the sentiment of movie reviews.

The next step in our model is to use *pooling* (specifically *max pooling*) on the output of the convolutional layers. This is similar to the FastText model where we performed the average over each of the word vectors, implemented by the F.avg_pool2d function, however instead of taking the average over a dimension, we are taking the maximum value over a dimension. Below an example of taking the maximum value (0.9) from the output of the convolutional layer on the example sentence (not shown is the activation function applied to the output of the convolutions).



The idea here is that the maximum value is the "most important" feature for determining the sentiment of the review, which corresponds to the "most important" n-gram within the review. How do we know what the "most important" n-gram is? Luckily, we don't have to! Through backpropagation, the weights of the filters are changed so that whenever certain n-grams that are highly indicative of the sentiment are seen, the output of the filter is a "high" value. This "high" value then passes through the max pooling layer if it is the maximum value in the output.

As our model has 100 filters of 3 different sizes, that means we have 300 different n-grams the model thinks are important. We concatenate these together into a single vector and pass them through a linear layer to predict the sentiment. We can think of the weights of this linear layer as "weighing up the evidence" from each of the 300 n-grams and making a final decision.

IMPLEMENTATION DETAILS :

We implement the convolutional layers with nn.Conv2d. The in_channels argument is the number of "channels" in your image going into the convolutional layer. In actual images this is usually 3 (one channel for each of the red, blue and green channels), however, when using the text we only have a single channel, the text itself. The out_channels is the number of filters and the kernel_size is the

size of the filters. Each of our kernel_sizes is going to be **[n x emb_dim]** where n is the size of the n-grams.

In PyTorch, RNNs want the input with the batch dimension second, whereas CNN wants the batch dimension first. Thus, the first thing we do to our input is permuted it to make it the correct shape. We then pass the sentence through an embedding layer to get our embeddings. The second dimension of the input into a nn.Conv2d layer must be the channel dimension. As text technically does not have a channel dimension, we un squeeze our tensor to create one. This matches with our in_channels=1 in the initialization of our convolutional layers.

We then pass the tensors through the convolutional and pooling layers, using the ReLU activation function after the convolutional layers. Another nice feature of the pooling layers is that they handle sentences of different lengths. The size of the output of the convolutional layer is dependent on the size of the input to it, and different batches contain sentences of different lengths. Without the max pooling layer, the input to our linear layer would depend on the size of the input sentence (not what we want). One option to rectify this would be to trim/pad all sentences to the same length, however with the max pooling layer we always know the input to the linear layer will be the total number of filters. **T**here an exception to this if your sentence(s) is shorter than the largest filter used. You will then have to pad your sentences to the length of the largest filter. In the IMDb data there are no reviews only 5 words long so we don't have to worry about that, but you will if you are using your own data.

Finally, we perform dropout on the concatenated filter outputs and then pass them through a linear layer to make our predictions.

Currently, the CNN model can only use 3 different sized filters, but we can actually improve the code of our model to make it more generic and take any number of filters. We do this by placing all of our convolutional layers in an nn.ModuleList, a function used to hold a list of PyTorch nn.Modules. If we simply used a standard Python list, the modules within the list cannot be "seen" by any modules outside the list which will cause us some errors.

We can now pass an arbitrarily sized list of filter sizes and the list comprehension will create a convolutional layer for each of them. Then, in the forward method, we iterate through the list applying each convolutional layer to get a list of convolutional outputs, which we also feed through the max pooling in a list comprehension before concatenating together and passing through the dropout and linear layers.

We can also implement the above model using 1-dimensional convolutional layers, where the embedding dimension is the "depth" of the filter and the number of tokens in the sentence is the width. We'll run our tests in this notebook using the 2-dimensional convolutional model, but leave the implementation for the 1-dimensional model below for anyone interested.

TRAIN THE MODEL

```
Epoch: 01 | Epoch Time: 0m 11s
      Train Loss: 0.644 | Train Acc: 62.23%
```

```
       Val. Loss: 0.487 |  Val. Acc: 78.46%
Epoch: 02 | Epoch Time: 0m 11s
      Train Loss: 0.419 | Train Acc: 80.98%
       Val. Loss: 0.360 |  Val. Acc: 84.72%
Epoch: 03 | Epoch Time: 0m 11s
      Train Loss: 0.302 | Train Acc: 87.17%
       Val. Loss: 0.345 |  Val. Acc: 85.25%
Epoch: 04 | Epoch Time: 0m 11s
      Train Loss: 0.218 | Train Acc: 91.27%
       Val. Loss: 0.318 |  Val. Acc: 87.04%
Epoch: 05 | Epoch Time: 0m 11s
      Train Loss: 0.155 | Train Acc: 94.35%
       Val. Loss: 0.339 |  Val. Acc: 86.56%
```
Test Loss: 0.339 | Test Acc: 85.28%

The input sentence has to be at least as long as the largest filter height used. We modify our predict_sentiment function to also accept a minimum length argument. If the tokenized input sentence is less than min_len tokens, we append padding tokens (<pad>) to make it min_len tokens.

MULTI-CLASS SENTIMENT ANALYSIS :

We'll be performing classification on a dataset with 6 classes. Note that this dataset isn't actually a sentiment analysis dataset, it's a dataset of questions and the task is to classify what category the question belongs to. However, everything covered in this notebook applies to any dataset with examples that contain an input sequence belonging to one of the C classes.

Below, we set up the fields, and load the dataset. The first difference is that we do not need to set the dtype in the LABEL field. When doing a multi-class problem, PyTorch expects the labels to be numericalized LongTensors. The second difference is that we use TREC instead of IMDB to load the TREC dataset. The fine_grained argument allows us to use the fine-grained labels (of which there are 50 classes) or not (in which case they'll be 6 classes). You can change this how you please.

The 6 labels (for the non-fine-grained case) correspond to the 6 types of questions in the dataset:

- HUM for questions about humans
- ENTY for questions about entities
- DESC for questions asking you for a description
- NUM for questions where the answer is numerical
- LOC for questions where the answer is a location
- ABBR for questions asking about abbreviations

Our loss function (aka criterion). Before we used BCEWithLogitsLoss, however now we use CrossEntropyLoss. Without going into too much detail, CrossEntropyLoss performs a *softmax* function over our model outputs and the loss is given by the between that and the label.

Generally:

CrossEntropyLoss is used when our examples exclusively belong to one of C classes

BCEWithLogitsLoss is used when our examples exclusively belong to only 2 classes (0 and 1) and are also used in the case where our examples belong to between 0 and C classes (aka multilabel classification).

Before, we had a function that calculated accuracy in the binary label case, where we said if the value was over 0.5 then we would assume it is positive. In the case where we have more than 2 classes, our model outputs a C dimensional vector, where the value of each element is the belief that the example belongs to that class.

For example, in our labels we have: 'HUM' = 0, 'ENTY' = 1, 'DESC' = 2, 'NUM' = 3, 'LOC' = 4 and 'ABBR' = 5. If the output of our model was something like: **[5.1, 0.3, 0.1, 2.1, 0.2, 0.6]** this means that the model strongly believes the example belongs to class 0, a question about a human, and slightly believes the example belongs to class 3, a numerical question.

We calculate the accuracy by performing an argmax to get the index of the maximum value in the prediction for each element in the batch and then counting how many times this equals the actual label. We then average this across the batch. The training loop is similar to before, without the need to squeeze the model predictions as CrossEntropyLoss expects the input to be **[batch size, n classes]** and the label to be **[batch size]**.

TRAIN THE MODEL :

Epoch: 01 | Epoch Time: 0m 0s

Train Loss: 1.310 | Train Acc: 47.99%

Val. Loss: 0.947 | Val. Acc: 66.81%

Epoch: 02 | Epoch Time: 0m 0s

Train Loss: 0.869 | Train Acc: 69.09%

Val. Loss: 0.746 | Val. Acc: 74.18%

Epoch: 03 | Epoch Time: 0m 0s

Train Loss: 0.665 | Train Acc: 76.94%

Val. Loss: 0.627 | Val. Acc: 78.03%

Epoch: 04 | Epoch Time: 0m 0s

Train Loss: 0.503 | Train Acc: 83.42%

Val. Loss: 0.548 | Val. Acc: 79.73%

Epoch: 05 | Epoch Time: 0m 0s

Train Loss: 0.376 | Train Acc: 87.88%
Val. Loss: 0.506 | Val. Acc: 81.40%

Test Loss: 0.411 | Test Acc: 87.15%

WORD EMBEDDINGS :

We have very briefly covered how word embeddings (also known as word vectors) are used in the tutorials. In this appendix, we'll have a closer look at these embeddings and find some (hopefully) interesting results.

Embeddings transform a one-hot encoded vector (a vector that is 0 in elements except one, which is 1) into a much smaller dimension vector of real numbers. The one-hot encoded vector is also known as a *sparse vector*, whilst the real-valued vector is known as a *dense vector*.

The key concept in these word embeddings is that words that appear in similar *contexts* appear nearby in the vector space, i.e. the Euclidean distance between these two-word vectors is small. By context here, we mean the surrounding words. For example in the sentences "I purchased some items at the shop" and "I purchased some items at the store" the words 'shop' and 'store' appear in the same context and thus should be close together in vector space.

You may have also heard about *word2vec*. *word2vec* is an algorithm (actually a bunch of algorithms) that calculates word vectors from a corpus. In this appendix we use *GloVe* vectors, *GloVe* being another algorithm to calculate word vectors. If you want to know how *Glove* works to check the website here[18].

In PyTorch, we use word vectors with the nn.Embedding layer, which takes a ***[sentence length, batch size]*** tensor and transforms it into a ***[sentence length, batch size, embedding dimensions]*** tensor.

In tutorial 2 onwards, we also used pre-trained word embeddings (specifically the GloVe vectors) provided by TorchText. These embeddings have been trained on a gigantic corpus. We can use these pre-trained vectors within any of our models, with the idea that as they have already learned

the context of each word they will give us a better starting point for our word vectors. This usually leads to faster training time and/or improved accuracy.

In this appendix we won't be training any models, instead, we'll be looking at the word embeddings and finding a few interesting things about them

LOADING THE GLOVE VECTORS :

First, we'll load the GloVe vectors. The name field specifies what the vectors have been trained on, here the 6B means a corpus of 6 billion words. The dim argument specifies the dimensionality of the word vectors. GloVe vectors are available in 50, 100, 200 and 300 dimensions. There is also a 42B and 840B glove vectors, however, they are only available at 300 dimensions. As shown above, there are 400,000 unique words in the GloVe vocabulary. These are the most common words found in the corpus the vectors were trained on. **In these set of GloVe vectors, every single word is lower-case only.**

SIMILAR CONTEXTS :

If we want to find the words similar to a certain input word, we first find the vector of this input word, then we scan through our vocabulary calculating the distance between the vector of each word and our input word vector. We then sort these from closest to furthest away.

man is to king as woman is to...

(4.0811) queen

(4.6429) monarch

(4.9055) throne

(4.9216) elizabeth

(4.9811) prince

This is the canonical example which shows off this property of word embeddings. So why does it work? Why does the vector of 'woman' added to the vector of 'king' minus the vector of 'man' give us 'queen'?

If we think about it, the vector calculated from 'king' minus 'man' gives us a "royalty vector". This is the vector associated with travelling from a man to his royal counterpart, a king. If we add this "royalty vector" to 'woman', this should travel to her royal equivalent, which is a queen!

CORRECTING SPELLING MISTAKES :

We'll put their findings into code and briefly explain them, but to read more about this, check out the original thread[24] and the associated write-up[25].

First, we need to load up the much larger vocabulary GloVe vectors, this is due to the spelling mistakes not appearing in the smaller vocabulary.

(0.0000) relieable

(5.0366) relyable

(5.2610) realible

(5.4719) realiable

(5.5402) relable

(5.5917) relaible

(5.6412) reliabe

(5.8802) relaiable

(5.9593) stabel

(5.9981) consitant

Notice how the correct spelling, "reliable", does not appear in the top 10 closest words. Surely the misspellings of a word should appear next to the correct spelling of the word as they appear in the same context, right?

The hypothesis is that misspellings of words are all equally shifted away from their correct spelling. This is because articles of text that contain spelling mistakes are usually written in an informal manner where correct spelling doesn't matter as much (such as tweets/blog posts), thus spelling errors will appear together as they appear in the context of informal articles.

Similar to how we created analogies before, we can create a "correct spelling" vector. This time, instead of using a single example to create our vector, we'll use the average of multiple examples. This will hopefully give better accuracy!

We first create a vector for the correct spelling, 'reliable', then calculate the difference between the "reliable vector" and each of the 8 misspellings of 'reliable'. As we are going to concatenate these 8 misspelling tensors together we need to unsqueeze a "batch" dimension to them.

EMBEDDINGS :

LOADING CUSTOM EMBEDDINGS:

Your embeddings need to be formatted so each line starts with the word followed by the values of the embedding vector, all space separated. All vectors need to have the same number of elements.

We create a Vector object by passing it the location of the embeddings (name), a location for the cached embeddings (cache) and a function to initialize tokens in our dataset that aren't within our embeddings (unk_init). As have done in previous notebooks, we have initialized these to N(0 ,1).

```
tensor([[ 1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
          1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
          1.0000,  1.0000,  1.0000,  1.0000],
        [ 1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
          1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
          1.0000,  1.0000,  1.0000,  1.0000],
        [ 1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
          1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
          1.0000,  1.0000,  1.0000,  1.0000],
        [-1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
         -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
         -1.0000, -1.0000, -1.0000, -1.0000],
        [-1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
         -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
         -1.0000, -1.0000, -1.0000, -1.0000],
        [-1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
         -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
         -1.0000, -1.0000, -1.0000, -1.0000],
        [ 0.5000, -0.5000,  0.5000, -0.5000,  0.5000, -0.5000,  0.5000, -0.5000,
          0.5000, -0.5000,  0.5000, -0.5000,  0.5000, -0.5000,  0.5000, -0.5000,
          0.5000, -0.5000,  0.5000, -0.5000]])
```

We then build our vocabulary, passing our Vectors object.

FREEZING AND UNFREEZING EMBEDDINGS:

We're going to train our model for 10 epochs. During the first 5 epochs, we are going to freeze the weights (parameters) of our embedding layer. For the last 10 epochs, we'll allow our embeddings to be trained.

Why would we ever want to do this? Sometimes the pre-trained word embeddings we use will already be good enough and won't need to be fine-tuned with our model. If we keep the embeddings frozen then we don't have to calculate the gradients and update the weights for these parameters, giving us faster training times. This doesn't really apply for the model used here, but we're mainly covering it to show how it's done. Another reason is that if our model has a large of parameters it may make training difficult, so by freezing our pre-trained embeddings we reduce the number of parameters needing to be learned.

To freeze the embedding weights, we set model.embedding.weight.requires_grad to False. This will cause no gradients to be calculated for the weights in the embedding layer, and thus no parameters will be updated when optimizer.step() is called. Then, during training, we check if FREEZE_FOR (which we set to 5) epochs have passed. If they have then we set model.embedding.weight.requires_grad to True, telling PyTorch that we should calculate gradients in the embedding layer and update them with our optimizer.

Epoch: 01 | Epoch Time: 0m 8s | Frozen? True

Train Loss: 0.729 | Train Acc: 52.97%

Val. Loss: 0.664 | Val. Acc: 57.37%

Epoch: 02 | Epoch Time: 0m 7s | Frozen? True

Train Loss: 0.668 | Train Acc: 59.46%

Val. Loss: 0.628 | Val. Acc: 67.05%

Epoch: 03 | Epoch Time: 0m 6s | Frozen? True

Train Loss: 0.641 | Train Acc: 62.80%

Val. Loss: 0.601 | Val. Acc: 68.72%

Epoch: 04 | Epoch Time: 0m 7s | Frozen? True

Train Loss: 0.614 | Train Acc: 66.05%

Val. Loss: 0.569 | Val. Acc: 72.59%

Epoch: 05 | Epoch Time: 0m 7s | Frozen? True

Train Loss: 0.598 | Train Acc: 67.40%

Val. Loss: 0.566 | Val. Acc: 71.26%

Epoch: 06 | Epoch Time: 0m 7s | Frozen? False

Train Loss: 0.573 | Train Acc: 69.34%

Val. Loss: 0.513 | Val. Acc: 76.40%

Epoch: 07 | Epoch Time: 0m 7s | Frozen? False

Train Loss: 0.541 | Train Acc: 72.60%

Val. Loss: 0.485 | Val. Acc: 77.91%

Epoch: 08 | Epoch Time: 0m 7s | Frozen? False

Train Loss: 0.511 | Train Acc: 75.07%

Val. Loss: 0.452 | Val. Acc: 79.36%

Epoch: 09 | Epoch Time: 0m 7s | Frozen? False

Train Loss: 0.471 | Train Acc: 77.63%

Val. Loss: 0.419 | Val. Acc: 81.24%

Epoch: 10 | Epoch Time: 0m 7s | Frozen? False

Train Loss: 0.430 | Train Acc: 80.04%

Val. Loss: 0.391 | Val. Acc: 82.51%

SAVING EMBEDDINGS:

We might want to re-use the embeddings we have trained here with another model. To do this, we'll write a function that will loop through our vocabulary, getting the word and embedding for each word, writing them to a text file in the same format as our custom embeddings so they can be used with TorchText again.

TORCHTEXT USING DATASETS:

READING JSON:

Starting with json, your data must be in the json lines format, i.e. it must be something like:

{"name": "John", "location": "United Kingdom", "age": 42, "quote": ["i", "love", "the", "united kingdom"]}

{"name": "Mary", "location": "United States", "age": 36, "quote": ["i", "want", "more", "telescopes"]}

That is, each line is a json object. See data/train.json for an example.

For json data, we must create a dictionary where:

- the key matches the key of the json object
- the value is a tuple where:
    - the first element becomes the batch object's attribute name
    - the second element is the name of the Field

What do we mean when we say "becomes the batch object's attribute name"? Recall in the previous exercises where we accessed the TEXT and LABEL fields in the train/evaluation loop by using batch.text and batch.label. Here, to access the name we use batch.n, to access the location we use batch.p, etc.

A few notes:

- The order of the keys in the fields dictionary does not matter, as long as its keys match the json data keys.
- The Field name does not have to match the key in the json object, e.g. we use PLACE for the "location" field.
- When dealing with json data, not all of the keys have to be used, e.g. we did not use the "age" field.
- Also, if the values of json field are a string then the Fields tokenization is applied (default is to split the string on spaces), however if the values are a list then no tokenization is applied. Usually it is a good idea for the data to already be tokenized into a list, this saves time as you don't have to wait for TorchText to do it.
- The value of the json fields do not have to be the same type. Some examples can have their "quote" as a string, and some as a list. The tokenization will only get applied to the ones with their "quote" as a string.
- If you are using a json field, every single example must have an instance of that field, e.g. in this example all examples must have a name, location and quote. However, as we are not using the age field, it does not matter if an example does not have it.

The path argument specifices the top level folder common among both datasets, and the train and test arguments specify the filename of each dataset, e.g. here the train dataset is located at data/train.json.

Also notice how the word "United Kingdom" in p has been split by the tokenization, whereas the "united kingdom" in s has not. This is due to what was mentioned previously, where TorchText assumes that any json fields that are lists are already tokenized and no further tokenization is applied.

READING CSV/TSV:

csv and tsv are very similar, except csv has elements separated by commas and tsv by tabs.

Using the same example above, our tsv data will be in the form of:

name location age quote

John United Kingdom 42 i love the united kingdom

Mary United States 36 i want more telescopes

That is, on each row the elements are separated by tabs and we have one example per row. The first row is usually a header (i.e. the name of each of the columns), but your data could have no header.You cannot have lists within tsv or csv data.

The way the fields are defined is a bit different to json. We now use a list of tuples, where each element is also a tuple. The first element of these inner tuples will become the batch object's attribute name, second element is the Field name. Unlike the json data, the tuples have to be in the same order that they are within the tsv data. Due to this, when skipping a column of data a tuple of Nones needs to be used, if not then our SAYING field will be applied to the age column of the tsv data and the quote column will not be used.

However, if you only wanted to use the name and age column, you could just use two tuples as they are the first two columns. We change our Tabular Dataset to read the correct .tsv files, and change the format argument to 'tsv'.If your data has a header, which ours does, it must be skipped by passing skip_header = True. If not, TorchText will think the header is an example. By default, skip_header will be False.

JSON OVER CSV/TSV :

Your csv or tsv data cannot be stored lists. This means data cannot be already be tokenized, thus everytime you run your Python script that reads this data via TorchText, it has to be tokenized. Using advanced tokenizers, such as the spaCy tokenizer, takes a non-negligible amount of time. Thus, it is better to tokenize your datasets and store them in the json lines format.

If tabs appear in your tsv data, or commas appear in your csv data, TorchText will think they are delimiters between columns. This will cause your data to be parsed incorrectly. Worst of all TorchText will not alert you to this as it cannot tell the difference between a tab/comma in a field and a tab/comma as a delimiter. As json data is essentially a dictionary, you access the data within the fields via its key, so do not have to worry about "surprise" delimiters.

ITERATORS :

By default, the train data is shuffled each epoch, but the validation/test data is sorted. However, TorchText doesn't know what to use to sort our data and it would throw an error if we don't tell it. There are two ways to handle this, you can either tell the iterator not to sort the validation/test data by passing sort = False, or you can tell it how to sort the data by passing a sort_key. A sort key is a function that returns a key on which to sort the data on. For example, lambda x: x.s will sort the examples by their s attribute, i.e their quote. Ideally, you want to use a sort key as the BucketIterator will then be able to sort your examples and then minimize the amount of padding within each batch. We can then iterate over our iterator to get batches of data. Note how by default TorchText has the batch dimension second.