**Sentiment Classification with Deep Learning**

**Adharsh Viswanath**Erik Jonsson School of Engineering & Computer Science  
The University of Texas at Dallas  
Richardson, TX 75080 USA  
[{viswanath}axv180007@utdallas.edu](mailto:%7bviswanath%7daxv180007@utdallas.edu)

**Abstract**

Knowledge about what the consumers feel about the product is invaluable in today’s corporate world. Thanks to the development of deep neural networks, the extraction of information contained in thousands of feedbacks can be done relatively quickly. In this project, I have tried to classify movie reviews based on their sentiment (positive or negative) using deep neural networks. Four neural network models are presented in this project – Baseline, Recurrent Neural Network (RNN, GRU and LSTM), Self-Attention and Convolutional Neural Network. The contextual meaning of words are represented through a 300 dimensional vector contained in the “wiki-news-300d-1M.vec” file and a dataset of 2000 movie reviews from the IMDB movie reviews dataset is used for training, validation and testing.

**Introduction**

Neural networks are an essential computational tool for language processing, and a very old one. A modern neural network is a network of small computing units, each of which takes a vector of input values and produces a single output value. In this project we introduce the neural net applied to classification. We introduce feed-forward network because the computation proceeds iteratively from one layer of units to the next. The use of modern neural nets is often called deep learning, because modern networks are often deep (have many layers).

Neural networks share much of the same mathematics as logistic regression. But neural networks are a more powerful classifier than logistic regression, and indeed a minimal neural network (technically one with a single ‘hidden layer’) can be shown to learn any function.

Neural net classifiers are different from logistic regression in another way. With logistic regression, we applied the regression classifier to many different tasks by developing many rich kinds of feature templates based on domain knowledge. When working with neural networks, it is more common to avoid the use of rich hand derived features, instead building neural networks that take raw words as inputs and learn to induce features as part of the process of learning to classify. Nets that are very deep are particularly good at representation learning for that reason deep neural nets are the right tool for large scale problems that offer sufficient data to learn features automatically.

Recurrent neural networks, long short-term memory and gated recurrent neural networks in particular, have been firmly established as state of the art approaches in sequence modeling and transduction problems such as language modeling and machine translation. Numerous efforts have since continued to push the boundaries of recurrent language models and encoder-decoder architectures. Recurrent models typically factor computation along the symbol positions of the input and output sequences. Aligning the positions to steps in computation time, they generate a sequence of hidden states ht, as a function of the previous hidden state ht−1 and the input for position t. This inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples. The fundamental constraint of sequential computation, however, remains. Attention mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks, allowing modeling of dependencies without regard to their distance in the input or output sequences. In all but a few cases, however, such attention mechanisms are used in conjunction with a recurrent network. This project accomplishes a very simple binary classification problem that identifies whether the sentiment associated with a chunk of text is positive or negative.

**Data**

The dataset used is the IMDB movie reviews dataset consisting of 2000 online movie reviews on the imdb website. The entire corpus is divided based on their true labels into two directories – pos (positive) and neg (negative) and is finally wrapped in a tar file – “movie\_reviews.tar”. The reviews are in the form of .txt files and the language is English.

**Methodology**

In this project, four models are considered – starting with a BaseSentiment model with just the embedding and fully connected layer, RNNSentiment model with an added RNN (Recurrent Neural Network) layer (VANILLA\_RNN, GRU or LSTM), AttentionSentiment model which replaces the RNN layer of the RNNSentiment model with a self-attention layer and finally the CNNSentiment which uses a CNN (Convolutional Neural Network) in between the embedding and fully connected layers.

**Word embedding**

First step of NLP is to convert words to vectors. There exist several methods for this task such as word2vec, the global vectors for word representation method (GloVe), fastText, and others. Among the pre-trained word vectors provided by word2vec, we choose the one created from Wikipedia corpus, which contains 1M words represented by vectors in 300 dimensions.

**Polarity**

We define the polarity (i.e., positive or negative) of reviews based on the words that appear in it. The embedding layer adds the word embedding dimension (=300) to each sample representation and this is what is taken as input by the subsequent layers. The first neural network layer tries to find new features from this input.

**Loss function**

We employ the nn.BCELoss() which is the Binary Cross Entropy Loss as our loss function since this is a simple binary classifier.

**Optimizer**

We use the optim.SGD() as our optimizer in this project. This uses the Stochastic Gradient Descent method to update the weights of the model under training.

**CUDA**

All models and training, validation and test batches of data were converted to cuda datatype to employ the power of GPU for faster computation.

**Loading the data**

This is divided further into five tasks:

**1. read file(path to dataset):** This method takes in a tar file and produces two lists, one containing the movie reviews and another containing classes.

For example, if the classes are positive and negative, this method outputs two lists as shown:

DATA = [\This movie sucks", \Good movie"]

LABELS = ["NEGATIVE", "POSITIVE"]

For this, we create two lists and fills the data list with the contents of a review text file and adds a corresponding true label entry to the labels list.

**2. preprocess(text):** This method creates a vocabulary and associates the tokens with a unique integer. For the previous example, the dictionary will be:

{'this': 1, 'movie': 2, 'sucks': 3, 'good': 4}

Here, instead of randomly assigning unique autoincrement integers to the tokens, we assign lower integer values to more frequent tokens for the sake of efficiency. In most of the NLP tasks, you will create an index mapping dictionary in such a way that your frequently occurring words are assigned lower indexes. One of the most common way of doing this is to use Counter method from Collections library.

**3. encode review(vocab, text):** Returns integer-encoded movie reviews. For the previous example, the reviews will now look like:

DATA = [[1, 2, 3], [4, 2]]

**4. encode labels(labels):** Here you integer-encode the labels (0 for NEGATIVE and 1 for POSITIVE) if you did not do that while reading the files. Labels will look like:

LABELS = [0, 1]

**5. pad zeros(encoded reviews, seq length):** Here you pad zeros to each review or truncate the review to make all reviews have constant length. For the previous example, lets say you want seq length to be 5. Then reviews will look:

DATA = [[1, 2, 3, 0, 0], [4, 2, 0, 0, 0]]

In this project, we have chosen the sequence length to be 200.

**Build the Embedding dictionary**

For this task you will load the pre-trained embedding vectors from Word2Vec. Specifically, you will associate each token id with its pre-trained vector.

For example, if the word2vec file contains these vectors:

this: 0.001 0.102 -1.221 movie: 9.011 -0.119 2.112 the: 1.223 0.911 -2.113

And the dictionary you obtained from the previous step was {'this': 1, 'movie': 2, 'sucks': 3, 'good': 4}

Then your embedding dict should like:

{1: tensor[0.001 0.102 -1.221], 2: tensor[9.011 -0.119 2.112], 3: tensor[0.0 0.0 0.0] 4: tensor[0.0 0.0 0.0]}

Note that the tensors associated with words sucks and good are zeros because the embedding file does not have embedding vectors for these tokens.

**Create a TensorDataset and DataLoader**

Once we have got our data in nice shape, we will split it into training, validation and test sets

train= 80% | valid = 10% | test = 10%

After creating our training, test and validation data. Next step is to create dataloaders for this data. We can use generator function for batching our data into batches instead we will use a TensorDataset. This is one of a very useful utility in PyTorch for using our data with DataLoaders with exact same ease as of torchvision datasets.

**BaseSentiment model**

Create the baseline deep learning model that has a very simple architecture. The model implements two layers:

**1. Embedding layer:** This takes in an input sample and represents it as the word embeddings obtained from the previous task.

**2. FCN:** A fully connected layer is added on top of the Embedding layer that gives you the class value associated with the input.

We are starting off with dataloaders (where we have defined batch\_size=32 and sequence length=200).

From the shape of input we can see that it is a tensor of 32 rows (=batch size) & 200 columns (=sequence length). This assures that our process of tokenization is working fine. This input will go as input into Embedding layer.

The module that allows you to use Embedding is torch.nn.Embedding. It takes two parameters : the vocabulary size and the dimensionality of the embedding. From the output of embedding layer we can see it has created a 3 dimensional tensor as a result of embedding weights. Now it has 50 rows, 200 columns and 300 embedding dimension i.e. for each tokenized word in our review we have added embedding dimension.

Note that before putting the embedding layer output into fc layer it has to be flattened out.

**Sigmoid Activation Layer:** This is needed just to turn all output value from fully connected layer into a value between 0 and 1.

This includes two steps: First, to reshape the output such that rows = batch size. Second, as we see in the Network Architecture — we only want the output after the last sequence (after the last timestep).

Observations: (Default: batch\_size = 32, learning rate = 0.01, n\_epochs = 30)

# batch\_size = 16:

# Training accuracy: 0.624, Validation accuracy: 0.510, Testing accuracy: 0.550

# batch\_size = 32:

# Training accuracy: 0.609, Validation accuracy: 0.600, Testing accuracy: 0.505

# batch\_size = 50:

# Training accuracy: 0.601, Validation accuracy: 0.450, Testing accuracy: 0.485

There was a decrease in training accuracy as the batch size was increased. The observations suggest that batch size of 32 is the best among the three.

# learning rate = 0.01:

# Training accuracy: 0.609, Validation accuracy: 0.600, Testing accuracy: 0.505

# learning rate = 0.03:

# Training accuracy: 0.624, Validation accuracy: 0.555, Testing accuracy: 0.565

# learning rate = 0.05:

# Training accuracy: 0.614, Validation accuracy: 0.480, Testing accuracy: 0.545

The increase in learning rate caused a decline in validation accuracy while pushing up the training accuracy. This suggests overfitting. Learning rate of 0.03 seems optimal.

# n\_epochs = 30:

# Training accuracy: 0.609, Validation accuracy: 0.600, Testing accuracy: 0.505

# n\_epochs = 60:

# Training accuracy: 0.635, Validation accuracy: 0.535, Testing accuracy: 0.445

# n\_epochs = 90:

# Training accuracy: 0.657, Validation accuracy: 0.500, Testing accuracy: 0.465

There is an increase in training accuracy with a corresponding decline in validation accuracy. Again, overfitting. 30 epochs give max testing accuracy.

Best combination: batch\_size = 16, learning rate = 0.03, n\_epochs = 30.

**RNNSentiment model**

Add an RNN layer to the baseline model in between the embedding layer and the FCN, add an RNN layer. Experimented with the following hyper-parameters in the RNN layer:

Type of RNN used: Vanilla RNN vs GRU vs LSTM

Number of RNN layers: The effect of increasing number of RNN

layers from 1 to 2.

Directionality: The effect of making the RNN layer bi-directional.

Observations: (Default: batch\_size = 32, learning rate = 0.01, n\_epochs = 30)

**Vanilla RNN:**

# batch\_size = 16:

# Training accuracy: 0.816, Validation accuracy: 0.550, Testing accuracy: 0.565

# batch\_size = 32:

# Training accuracy: 0.734, Validation accuracy: 0.575, Testing accuracy: 0.515

# batch\_size = 50:

# Training accuracy: 0.666, Validation accuracy: 0.505, Testing accuracy: 0.485

# bidirectional: Training accuracy: 0.616, Validation accuracy: 0.545, Testing accuracy: 0.490

# learning rate = 0.01:

# Training accuracy: 0.734, Validation accuracy: 0.575, Testing accuracy: 0.515

# learning rate = 0.03:

# Training accuracy: 0.862, Validation accuracy: 0.510, Testing accuracy: 0.555

# learning rate = 0.05:

# Training accuracy: 0.865, Validation accuracy: 0.560, Testing accuracy: 0.510

# n\_epochs = 30:

# Training accuracy: 0.734, Validation accuracy: 0.575, Testing accuracy: 0.515

# n\_epochs = 60:

# Training accuracy: 0.876, Validation accuracy: 0.530, Testing accuracy: 0.475

# n\_epochs = 90:

# Training accuracy: 0.901, Validation accuracy: 0.530, Testing accuracy: 0.445

# number of layers = 2:

# Training accuracy: 0.821, Validation accuracy: 0.475, Testing accuracy: 0.525

Best combination: batch\_size = 16, learning rate = 0.03, n\_epochs = 30, number of layers = 2, bidirectional.

**GRU:**

# batch\_size = 16:

# Training accuracy: 0.642, Validation accuracy: 0.555, Testing accuracy: 0.565

# batch\_size = 32:

# Training accuracy: 0.614, Validation accuracy: 0.455, Testing accuracy: 0.475

# batch\_size = 50:

# Training accuracy: 0.591, Validation accuracy: 0.510, Testing accuracy: 0.500

# bidirectional: Training accuracy: 0.584, Validation accuracy: 0.545, Testing accuracy: 0.510

# learning rate = 0.01:

# Training accuracy: 0.614, Validation accuracy: 0.455, Testing accuracy: 0.475

# learning rate = 0.03:

# Training accuracy: 0.691, Validation accuracy: 0.525, Testing accuracy: 0.530

# learning rate = 0.05:

# Training accuracy: 0.752, Validation accuracy: 0.525, Testing accuracy: 0.530

# n\_epochs = 30:

# Training accuracy: 0.614, Validation accuracy: 0.455, Testing accuracy: 0.475

# n\_epochs = 60:

# Training accuracy: 0.683, Validation accuracy: 0.495, Testing accuracy: 0.505

# n\_epochs = 90:

# Training accuracy: 0.702, Validation accuracy: 0.520, Testing accuracy: 0.530

# number of layers = 2:

# Training accuracy: 0.551, Validation accuracy: 0.520, Testing accuracy: 0.475

Best combination: batch\_size = 16, learning rate = 0.03, n\_epochs = 90, number of layers = 1, bidirectional.

**LSTM:**

# batch\_size = 16:

# Training accuracy: 0.608, Validation accuracy: 0.545, Testing accuracy: 0.510

# batch\_size = 32:

# Training accuracy: 0.549, Validation accuracy: 0.475, Testing accuracy: 0.470

# batch\_size = 50:

# Training accuracy: 0.557, Validation accuracy: 0.500, Testing accuracy: 0.465

# bidirectional: Training accuracy: 0.534, Validation accuracy: 0.525, Testing accuracy: 0.540

# learning rate = 0.01:

# Training accuracy: 0.549, Validation accuracy: 0.475, Testing accuracy: 0.470

# learning rate = 0.03:

# Training accuracy: 0.616, Validation accuracy: 0.490, Testing accuracy: 0.520

# learning rate = 0.05:

# Training accuracy: 0.710, Validation accuracy: 0.455, Testing accuracy: 0.495

# n\_epochs = 30:

# Training accuracy: 0.549, Validation accuracy: 0.475, Testing accuracy: 0.470

# n\_epochs = 60:

# Training accuracy: 0.581, Validation accuracy: 0.540, Testing accuracy: 0.475

# n\_epochs = 90:

# Training accuracy: 0.643, Validation accuracy: 0.550, Testing accuracy: 0.480

# number of layers = 2:

# Training accuracy: 0.528, Validation accuracy: 0.510, Testing accuracy: 0.465

Best combination: batch\_size = 16, learning rate = 0.03, n\_epochs = 90, number of layers = 1, bidirectional.

**AttentionSentiment model**

Replace the RNN implemented in the previous model by self-attention.

With batch\_size = 32, learning rate = 0.01, n\_epochs = 30, the results were:

# Training accuracy: 0.648, Validation accuracy: 0.640, Testing accuracy: 0.535

The accuracy was noticeably improved compared to previous models.

Observations: (Default: batch\_size = 32, learning rate = 0.01, n\_epochs = 30)

# batch\_size = 16:

# Training accuracy: 0.666, Validation accuracy: 0.570, Testing accuracy: 0.525

# batch\_size = 32:

# Training accuracy: 0.648, Validation accuracy: 0.640, Testing accuracy: 0.535

# batch\_size = 50:

# Training accuracy: 0.638, Validation accuracy: 0.620, Testing accuracy: 0.615

# learning rate = 0.01:

# Training accuracy: 0.648, Validation accuracy: 0.640, Testing accuracy: 0.535

# learning rate = 0.03:

# Training accuracy: 0.694, Validation accuracy: 0.595, Testing accuracy: 0.635

# learning rate = 0.05:

# Training accuracy: 0.509, Validation accuracy: 0.455, Testing accuracy: 0.470

# n\_epochs = 30:

# Training accuracy: 0.648, Validation accuracy: 0.640, Testing accuracy: 0.535

# n\_epochs = 60:

# Training accuracy: 0.686, Validation accuracy: 0.600, Testing accuracy: 0.615

# n\_epochs = 90:

# Training accuracy: 0.708, Validation accuracy: 0.560, Testing accuracy: 0.570

Best combination: batch\_size = 32, learning rate = 0.03, n\_epochs = 60.

**Experiments:**

**1) CNNSentiment model**

As an experiment, a model with Convolutional Neural Network layer in between embedding and fully connected layer was implemented.

The results for batch\_size = 32, learning rate = 0.01, n\_epochs = 30 were:

# Training accuracy = 0.531, Validation accuracy: 0.540, Testing accuracy: 0.475

Its performance were similar to RNN models in this case, except GRU.

It seems that the self-attention model is the most superior one for sentiment analysis.

2) When pretraining was applied to embedding layer weights, all models recorded higher testing accuracy except the self-attention model.

**Conclusion**

Through the experiments, it has been established that even though RNN models are good in sentiment analysis, self-attention trumps them all. It is still an ongoing research area, so we can expect more amazing results from this technique.

The use of cuda, efficient encoding of tokens and the use of numpy greatly contributed to the speed of computation. Overfitting was visible on almost all models except self-attention especially when run for more number of epochs. The use of dropouts did help in reducing this to an extent. The availability of IMDB movie review dataset and the word2vec embeddings greatly assisted me in this endeavor. The help of our professor Mr. Dan Moldovan and our T.A Takshak Desai was truly invaluable.

**References**

1. https://pytorch.org/docs/stable/index.html

2. https://towardsdatascience.com/sentiment-analysis-using-lstm-step-by-step-50d074f09948

3. https://colab.research.google.com/github/agungsantoso/deep-learning-v2-

pytorch/blob/master/sentiment-rnn/Sentiment RNN Exercise.ipynb

4. https://towardsdatascience.com/how-to-code-the-transformer-in-pytorch-

24db27c8f9ec

5. https://towardsdatascience.com/reading-between-the-layers-lstm-network-7956ad192e58

6. https://github.com/prakashpandey9/Text-Classification-Pytorch/tree/master/models

7. https://www.aiworkbox.com/lessons/how-to-define-a-convolutional-layer-in-pytorch