Using Deep Learning for End to End Multiclass Text Classification

Article by **Rahul Agarwal** | April 02, 2020

Have you ever thought about how toxic comments get flagged automatically on platforms like Quora or Reddit? Or how mail gets marked as spam? Or what decides which online ads are shown to you?

All of the above are examples of how text classification is used in different areas. Text classification is a common task in natural language processing (NLP) which transforms a sequence of text of indefinite length into a single category.

One theme that emerges from the above examples is that all have a binary target class. For example, either the comment is toxic or not toxic, or the review is fake or not fake. In short, there are only two target classes, hence the term binary.

But this is not always the case, and some problems might have more than two target classes. These problems are conveniently termed multiclass classifications, and it is these problems we'll focus on in this post. Some examples of multiclass classification include:
The sentiment of a review: positive, negative or neutral (three classes)
News Categorization by genre: Entertainment, education, politics, etc.
In this post, we will go through a multiclass text classification problem using various Deep Learning Methods.

Dataset / Problem Description

For this post I am using the UCI ML Drug Review dataset from Kaggle. It contains over 200,000 patient drug reviews, along with related conditions. The dataset has many columns, but we will be using just two of them for our NLP Task.

So, our dataset mostly looks like this:

Task: We want to classify the top disease conditions based on the drug review.

A Primer on word2vec embeddings:

Before we go any further into text classification, we need a way to represent words numerically in a vocabulary. Why? Because most of our ML models require numbers, not text.

One way to achieve this goal is by using one-hot encoding of word vectors, but this is not the right choice. Given a vast vocabulary, this representation would take a lot of space, and it cannot accurately express the similarity between different words, such as if we want to find the cosine similarity between numerical words x and y:

Given the structure of one-hot encoded vectors, the similarity is always going to be 0 between different words.

Word2Vec overcomes the above difficulties by providing us with a fixed-length (usually much smaller than the vocabulary size) vector representation of words. It also captures the similarity and analogous relationships between different words.

Word2vec vectors of words are learned in such a way that they allow us to learn different analogies. This enables us to do algebraic manipulations on words that were not possible previously.

For example: What is king – man + woman? The result is Queen.

Word2Vec vectors also help us to find the similarity between words. If we look for similar words to "good", we will find awesome, great, etc. It is this property of word2vec that makes it invaluable for text classification. **With this, our deep learning network understands that "good" and "great" are words with similar meanings.**

In simple terms, word2vec creates fixed-length vectors for words, giving us a d dimensional vector for every word (and common bigrams) in a dictionary.

These word vectors are usually pre-trained, and provided by others after training on a large corpora of texts like Wikipedia, Twitter, etc. The most commonly used pre-trained word vectors are Glove and Fast text with 300-dimensional word vectors. In this post, we will use the Glove word vectors.

Data Preprocessing

In most cases, text data is not entirely clean. Data coming from different sources have different characteristics, and this makes text preprocessing one of the most critical steps in the classification pipeline. For example, Text data from Twitter is different from the text data found on Quora or other news/blogging platforms, and each needs to be treated differently. However, the techniques we'll cover in this post are generic enough for almost any kind of data you might encounter in the jungles of NLP.

a) Cleaning Special Characters and Removing Punctuation

Our preprocessing pipeline depends heavily on the word2vec embeddings we are going to use for our classification task. **In principle, our preprocessing should match the preprocessing used before training the word embedding.** Since most of the embeddings don't provide vector values for punctuation and other special characters, the first thing we want to do is get rid of the special characters in our text data.

```
# Some preprocesssing that will be common to all the text classification methods you will see.
import re
def clean_text(x):
    pattern = r'[^a-zA-z0-9\s]'
    text = re.sub(pattern, '', x)
    return x
```

b) Cleaning Numbers

Why do we want to replace numbers with #s? Because most embeddings, including Glove, have preprocessed their text in this way.

Small Python Trick: We use an if statement in the code below to check beforehand if a number exists in a text because an if is always faster than a re.sub command, and most of our text doesn't contain numbers.

```
def clean_numbers(x):
    if bool(re.search(r'\d', x)):
        x = re.sub('[0-9]{5,}', '#####', x)
        x = re.sub('[0-9]{4}', '####', x)
        x = re.sub('[0-9]{3}', '###', x)
        x = re.sub('[0-9]{2}', '##', x)
    return x
```

c) Removing Contractions

Contractions are words that we write with an apostrophe. Examples of contractions are words like "ain't" or "aren't". Since we want to standardize our text, it makes sense to expand these contractions. Below we have done this using contraction mapping and regex functions.

```
contraction_dict = {"ain't": "is not", "aren't": "are not","can't": "cannot", "'cause": "because", "could've": "could have"}
def _get_contractions(contraction_dict):
    contraction_re = re.compile('(%s)' % '|'.join(contraction_dict.keys()))
    return contraction_dict, contraction_re
contractions, contractions_re = _get_contractions(contraction_dict)
def replace_contractions(text):
    def replace(match):
        return contractions[match.group(0)]
    return contractions_re.sub(replace, text)
# Usage
replace_contractions("this's a text with contraction")
```

Apart from the above techniques, you may want to do spell correction, too. But since our post is already quite long, we'll leave that for now.

Data Representation: Sequence Creation

One thing that has made deep learning a go-to choice for NLP is the fact that we don't have to hand-engineer features from our text data; deep learning algorithms take as input a sequence of text to learn its structure just like humans do. Since machines cannot understand words, they expect their data in numerical form. So we need to represent our text data as a series of numbers.

To understand how this is done, we need to understand a little about the Keras Tokenizer function. Other tokenizers are also viable, but the Keras Tokenizer is a good choice for me.

a) Tokenizer

Put simply, a tokenizer is a utility function that splits a sentence into words. `keras.preprocessing.text.Tokenizer` tokenizes (splits) a text into tokens (words) while keeping only the words that occur the most in the text corpus.

```
#Signature:
Tokenizer(num_words=None, filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n',
lower=True, split=' ', char_level=False, oov_token=None, document_count=0, **kwargs)
```

The num_words parameter keeps only a pre-specified number of words in the text. This is helpful because we don't want our model to get a lot of noise by considering words that occur infrequently. In real-world data, most of the words we leave using the num_words parameter are normally misspelled words. The tokenizer also filters some non-wanted tokens by default and converts the text into lowercase.

Once fitted to the data, the tokenizer also keeps an index of words (a dictionary we can use to assign unique numbers to words), which can be accessed by tokenizer.word_index. The words in the indexed dictionary are ranked in order of frequency.

So the whole code to use the tokenizer is as follows:

```
from keras.preprocessing.text import Tokenizer
## Tokenize the sentences
tokenizer = Tokenizer(num_words=max_features)
tokenizer.fit_on_texts(list(train_X)+list(test_X))
train_X = tokenizer.texts_to_sequences(train_X)
test_X = tokenizer.texts_to_sequences(test_X)
```

where `train_X` and `test_X` are lists of documents in the corpus.

b) Pad Sequence

Normally our model expects that each text sequence (each training example) will be of the same length (the same number of words/tokens). We can control this using the `maxlen` parameter.

For example:

```
train_X = pad_sequences(train_X, maxlen=maxlen)
test_X = pad_sequences(test_X, maxlen=maxlen)
```

Now our training data contains a list of numbers. Each list has the same length. And we also have the `word_index` which is a dictionary of the words that occur most in the text corpus.

c) Label Encoding the Target Variable

The Pytorch model expects the target variable as a number and not a string. We can use Label encoder from `sklearn` to convert our target variable.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
train_y = le.fit_transform(train_y.values)
test_y = le.transform(test_y.values)
```

Load Embedding

First, we need to load the required Glove embeddings.

```
def load_glove(word_index):
    EMBEDDING_FILE = 'data/glove.840B.300d.txt'
    def get_coefs(word,*arr): return word, np.asarray(arr, dtype='float32')[:300]
    embeddings_index = dict(get_coefs(*o.split(" ")) for o in open(EMBEDDING_FILE))

    all_embs = np.stack(embeddings_index.values())
    emb_mean,emb_std = -0.005838499,0.48782197
    embed_size = all_embs.shape[1]

nb_words = min(max_features, len(word_index)+1)
    embedding_matrix = np.random.normal(emb_mean, emb_std, (nb_words, embed_size))
    for word, i in word_index.items():
        if i >= max_features: continue
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector
        else:
            embedding_vector = embeddings_index.get(word.capitalize())
            if embedding_vector is not None:
                embedding_matrix[i] = embedding_vector
    return embedding_matrix

embedding_matrix = load_glove(tokenizer.word_index)
```

Be sure to put the path of the folder where you download these GLoVE vectors. What does the `embeddings_index` contain? It's a dictionary in which the key is the word, and the value is the word vector, a `np.array` of length 300. The length of this dictionary is somewhere around a billion.

Since we only want the embeddings of words that are in our `word_index`, we will create a matrix that just contains required embeddings using the word index from our tokenizer.

Deep Learning Models

1. TextCNN

The idea of using a CNN to classify text was first presented in the paper Convolutional Neural Networks for Sentence Classification by Yoon Kim.

Representation: The central concept of this idea is to **see our documents as images**. But how? Let's say we have a sentence, and we have maxlen = 70 and embedding size = 300. We can create a matrix of numbers with the shape 70×300 to represent this sentence. Images also have a matrix where individual elements are pixel values. But instead of image pixels, the input to the task is sentences or documents represented as a matrix. Each row of the matrix corresponds to a one-word vector.
Convolution Idea: For images, we move our conv. filter both horizontally as well as vertically, but for text we fix kernel size to filter_size x embed_size, i.e. (3,300) we are just going to move vertically down the convolution looking at three words at once, since our filter size in this case is 3. This idea seems right since our convolution filter is not splitting word embedding; it gets to look at the full embedding of each word. Also, one can think of filter sizes as unigrams, bigrams, trigrams, etc. Since we are looking at a context window of 1, 2, 3, and 5 words respectively.

Here is the text classification CNN network coded in Pytorch.

```
class CNN_Text(nn.Module):
    def __init__(self):
        super(CNN_Text, self).__init__()
        filter_sizes = [1,2,3,5]
        num_filters = 36
        n_classes = len(le.classes_)
        self.embedding = nn.Embedding(max_features, embed_size)
        self.embedding.weight = nn.Parameter(torch.tensor(embedding_matrix, dtype=torch.float32))
        self.embedding.weight.requires_grad = False
        self.convs1 = nn.ModuleList([nn.Conv2d(1, num_filters, (K, embed_size)) for K in filter_sizes])
        self.dropout = nn.Dropout(0.1)
        self.fc1 = nn.Linear(len(filter_sizes)*num_filters, n_classes)
def forward(self, x):
        x = self.embedding(x)
        x = x.unsqueeze(1)
        x = [F.relu(conv(x)).squeeze(3) for conv in self.convs1]
        x = [F.max_pool1d(i, i.size(2)).squeeze(2) for i in x]
        x = torch.cat(x, 1)
        x = self.dropout(x)
        logit = self.fc1(x)
        return logit
```

2. BiDirectional RNN (LSTM/GRU)

TextCNN works well for text classification because it takes care of words in close range. For example, it can see "new york" together. However, it still can't take care of all the context provided in a particular text sequence. It still does not learn the sequential structure of the data, where each word is dependent on the previous word, or a word in the previous sentence.

RNNs can help us with that. **They can remember previous information using hidden states and connect it to the current task.**

Long Short Term Memory networks (LSTM) are a subclass of RNN, specialized in remembering information for extended periods. Moreover, a bidirectional LSTM keeps the contextual information in both directions, which is pretty useful in text classification tasks (However, it won't work for a time series prediction task as we don't have visibility into the future in this case).

For a simple explanation of a bidirectional RNN, think of an RNN cell as a black box taking as input a hidden state (a vector) and a word vector and giving out an output vector and the next hidden state. This box has some weights which need to be tuned using backpropagation of the losses. Also, the same cell is applied to all the words so that the weights are shared across the words in the sentence. This phenomenon is called weight-sharing.

```
Hidden state, Word vector ->(RNN Cell) -> Output Vector , Next Hidden state
```

For a sequence of length 4 like **"you will never believe"**, The RNN cell gives 4 output vectors, which can be concatenated and then used as part of a dense feedforward architecture.

In the bidirectional RNN, the only change is that we read the text in the usual fashion as well in reverse. So we stack two RNNs in parallel, and we get 8 output vectors to append.

Once we get the output vectors, we send them through a series of dense layers and finally, a softmax layer to build a text classifier.

In most cases, you need to understand how to stack some layers in a neural network to get the best results. We can try out multiple bidirectional GRU/LSTM layers in the network if it performs better.

Due to the limitations of RNNs, such as not remembering long term dependencies, in practice we almost always use LSTM/GRU to model long term dependencies. In this case, you can think of the RNN cell being replaced by an LSTM cell or a GRU cell in the above figure.

Here is some code in Pytorch for this network:

```
class BiLSTM(nn.Module):

    def __init__(self):
        super(BiLSTM, self).__init__()
        self.hidden_size = 64
        drp = 0.1
        n_classes = len(le.classes_)
        self.embedding = nn.Embedding(max_features, embed_size)
        self.embedding.weight = nn.Parameter(torch.tensor(embedding_matrix, dtype=torch.float32))
        self.embedding.weight.requires_grad = False
        self.lstm = nn.LSTM(embed_size, self.hidden_size, bidirectional=True, batch_first=True)
        self.linear = nn.Linear(self.hidden_size*4 , 64)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(drp)
        self.out = nn.Linear(64, n_classes)
def forward(self, x):
        #rint(x.size())
        h_embedding = self.embedding(x)
        #_embedding = torch.squeeze(torch.unsqueeze(h_embedding, 0))
        h_lstm, _ = self.lstm(h_embedding)
        avg_pool = torch.mean(h_lstm, 1)
        max_pool, _ = torch.max(h_lstm, 1)
        conc = torch.cat(( avg_pool, max_pool), 1)
        conc = self.relu(self.linear(conc))
        conc = self.dropout(conc)
        out = self.out(conc)
        return out
```

Training

Below is the code we use to train our BiLSTM Model. The code is well commented, so please go through the code to understand it. You might also want to look at my post on Pytorch.

```
n_epochs = 6
model = BiLSTM() #Use CNN_Text() for CNN Model
loss_fn = nn.CrossEntropyLoss(reduction='sum')
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=0.001)
model.cuda()
# Load train and test in CUDA Memory
x_train = torch.tensor(train_X, dtype=torch.long).cuda()
y_train = torch.tensor(train_y, dtype=torch.long).cuda()
x_cv = torch.tensor(test_X, dtype=torch.long).cuda()
y_cv = torch.tensor(test_y, dtype=torch.long).cuda()
# Create Torch datasets
train = torch.utils.data.TensorDataset(x_train, y_train)
valid = torch.utils.data.TensorDataset(x_cv, y_cv)
# Create Data Loaders
train_loader = torch.utils.data.DataLoader(train, batch_size=batch_size, shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid, batch_size=batch_size, shuffle=False)
train_loss = []
valid_loss = []
for epoch in range(n_epochs):
    start_time = time.time()
```

```
# Set model to train configuration
model.train()
avg_loss = 0.
    for i, (x_batch, y_batch) in enumerate(train_loader):
        # Predict/Forward Pass
        y_pred = model(x_batch)
        # Compute loss
        loss = loss_fn(y_pred, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        avg_loss += loss.item() / len(train_loader)

    # Set model to validation configuration - Doesn't get trained here
    model.eval()
    avg_val_loss = 0.
    val_preds = np.zeros((len(x_cv),len(le.classes_)))

    for i, (x_batch, y_batch) in enumerate(valid_loader):
        y_pred = model(x_batch).detach()
        avg_val_loss += loss_fn(y_pred, y_batch).item() / len(valid_loader)
        # keep/store predictions
        val_preds[i * batch_size:(i+1) * batch_size] =F.softmax(y_pred).cpu().numpy()

    # Check Accuracy
    val_accuracy = sum(val_preds.argmax(axis=1)==test_y)/len(test_y)
    train_loss.append(avg_loss)
    valid_loss.append(avg_val_loss)
    elapsed_time = time.time() - start_time
    print('Epoch {}/{} \t loss={:.4f} \t val_loss={:.4f} \t val_acc={:.4f} \t time={:.2f}s'.format(
            epoch + 1, n_epochs, avg_loss, avg_val_loss, val_accuracy, elapsed_time))
```

The training output looks like below:

Results/Prediction

Below is the confusion matrix for the results of the BiLSTM model. We can see that our model does reasonably well, with an 87% accuracy on the validation dataset.

What's interesting is that even at points where the model performs poorly, it is quite understandable. For example, the model gets confused between weight loss and obesity, or between depression and anxiety, or between depression and bipolar disorder. I am not an expert, but these diseases do feel quite similar.

```
import scikitplot as skplt
y_true = [le.classes_[x] for x in test_y]
y_pred = [le.classes_[x] for x in val_preds.argmax(axis=1)]
skplt.metrics.plot_confusion_matrix(
    y_true,
    y_pred,
    figsize=(12,12),x_tick_rotation=90)
```

Conclusion

In this post, we covered deep learning architectures like LSTM and CNN for text classification, and explained the different steps used in deep learning for NLP.

There is still a lot that can be done to improve this model's performance. Changing the learning rates, using learning rate schedules, using extra features, enriching embeddings, removing misspellings, etc. I hope this boilerplate code provides a go-to baseline for any text classification problem you might face.

You can find the full working code here on Github, or this Kaggle Kernel.

Also, if you want to learn more about NLP, here is an excellent course.
Share this post:

Subscribe to our newsletter for more technical articles

Subscribe

The Author

Rahul Agarwal

Rahul is a data scientist currently working with WalmartLabs. He enjoys working with data-intensive problems and is constantly in search of new ideas to work on. Contact him on Twitter: @MLWhiz

Related resources

5 Best Image Annotation Companies for Big Data Outsourcing
20 Best Instagram Accounts for AI Enthusiasts to Follow
The Ethical Debate on AI Applications: An Interview with Data Scientist Dr. Iain Brown
7 must-read Quora answers about how to build machine learning models
What are Search & Recommendation Systems in Machine Learning?
Companies Hiring Remote Workers During COVID-19
What is Computer Vision?
12 Best Outsourced Data Entry Services for Machine Learning
Top 10 Text Labeling Services for Machine Learning
5 Strategic Uses for AI in Ecommerce
What is Data Annotation and How is it Used in Machine Learning?
Microsoft AirSim on Unity Streamlines Autonomous Vehicle Training