It is the effort that counts?

Introduction:

"If at first you don't succeed, redefine success to include effort." - Some famous guy

After writing hundreds of lines of code, going through a ton of documentation and sample codes, reading research papers to get to the depth of the idea, I was unable to train the model with a high enough accuracy. During my attempt to train a BERT model, I encountered challenges that led to low accuracy. Despite the suboptimal outcome, I invested a significant amount of effort to overcome these obstacles and make the model work. As a result of this process, I gained valuable knowledge and insight into the workings of all the steps involved in training a neural network and other techniques in NLP. Although the final accuracy of the model may not have met my initial expectations, the effort expended and knowledge acquired in the process of attempting to improve it was valuable. I consider this experience to be a positive one, as it served to expand my understanding of NLP and the challenges involved in working on it.

Dataset:

The dataset is a text dataset that has been created for the purpose of classification in the E-commerce domain. It is composed of approximately 27,000 samples, and has been divided into four categories: "Electronics", "Household", "Books", and "Clothing & Accessories". These categories have been chosen as they are representative of the most common types of products found on E-commerce websites, covering about 80% of the products sold online. The text data in this dataset is expected to contain information such as product descriptions, reviews, or specifications, which are typically used by customers to make purchasing decisions. By using this dataset, researchers and developers can train machine learning models to accurately classify products based on their textual descriptions, which can be a useful tool in improving the user experience and recommendation systems on E-commerce platforms. The dataset was picked up from Kaggle.

While trying to train the model, I decided to experiment with another dataset as well. This dataset contains approximately 210,000 news headlines published by HuffPost between the years 2012 and 2022. The dataset is one of the largest news datasets available and can be used as a benchmark for a range of computational linguistic tasks. The dataset comprises a diverse range of news topics, including politics, entertainment, business, and sports, among others. This was also picked up from Kaggle.

1. Introduction and Exploratory Data Analysis

To begin the exploratory data analysis phase of the project, I first imported the necessary libraries. Specifically, I used Pandas for data manipulation and analysis, NumPy for mathematical operations, and Matplotlib and Seaborn for data visualization. I used Pandas functions to examine the dimensions of the dataset, check for missing values, and drop any duplicates.I used Seaborn and Matplotlib to create various types of charts and plots to visualize the data and identify any patterns or trends that may exist.

2. Cleaning and Preprocessing

Here's a brief summary of what each function does:

1. convert_to_lowercase(text): This function takes a text string as input and converts it to lowercase.
2. remove_whitespace(text): This function takes a text string as input and removes any leading or trailing whitespace.

3. remove_punctuation(text): This function takes a text string as input and removes any punctuation from it. Apostrophes are kept intact to preserve contractions.
4. remove_html(text): This function takes a text string as input and removes any HTML tags from it.
5. remove_emoji(text): This function takes a text string as input and removes any emojis from it.
6. remove_http(text): This function takes a text string as input and removes any URLs or hyperlinks from it.
7. remove_stopwords(text): This function takes a text string as input and removes any stopwords (common words like "a", "an", "the", etc.) from it.
8. discard_non_alpha(text): This function takes a text string as input and discards any non-alphabetic characters (numbers, symbols, etc.).
9. text_lemmatizer(text): This function takes a text string as input and performs lemmatization on it using the Spacy library.

The following libraries were used in the text preprocessing code:

1. string: This is a built-in Python library that provides a collection of string constants and functions, including a string of all ASCII punctuation characters.
2. re: This is another built-in Python library that provides support for regular expressions, which are used to match patterns in strings.
3. nltk: This is the Natural Language Toolkit library, which provides a wide range of tools and resources for working with human language data in Python.
4. spacy: This is another popular library for natural language processing, providing tools for tokenization, lemmatization, named entity recognition, and more.

Next, we convert the categorical target variable to numerical value. The LabelEncoder class is a utility provided by scikit-learn. It is used for encoding categorical variables as integers, which can then be used as input for machine learning models.

In the context of supervised learning, where the objective is to predict an outcome variable based on a set of input variables, the outcome variable (or the target variable) is often categorical in nature. Many machine learning algorithms require numeric input data, so it is necessary to convert the categorical variable to numeric form.

3. Steps before training a model

The BertTokenizer class is used to tokenize text data, which is the process of breaking down a text document into smaller units, such as words or subwords, that can be fed into a machine learning model. The tokenizer is based on the BERT (Bidirectional Encoder Representations from Transformers) model, which will be described below.

The bert-base-uncased configuration means that the model is trained on a lowercased version of the English language. This is useful because it allows the model to generalize better to new text data, as it reduces the complexity of the input data and makes it easier to process.

The code truncates the line to a maximum length of 512 tokens using line[:512]. This is because the BERT model has a maximum sequence length of 512 tokens, and inputs longer than that will cause the model to run out of memory.

Next, the encode method of the tokenizer object is called on the truncated line, with add_special_tokens=True specified. This adds special tokens (such as [CLS] and [SEP]) to mark the beginning and end of the input text, which are required by the BERT model.

The code then splits the input ids, labels and attention masks into three sets: training set, validation set, and test set. It uses the train_test_split function from scikit-learn library to split the data randomly.

Initially, the input ids, labels and attention_masks are split into training set and remaining set using a 70-30 split ratio. The remaining set is further split into validation set and test set using a 50-50 split ratio.

The random_state parameter is set to 42 to ensure reproducibility of the split across different runs of the program.

Then, we use the PyTorch library to create a TensorDataset which contains the input data, the attention masks and the corresponding labels as tensors. Then, the DataLoader function is used to load the data in batches for efficient training. batch_size is a hyperparameter that defines the number of samples that will be propagated through the neural network at once. The shuffle parameter shuffles the data before each epoch to help generalize the model better.

Then the code imports the pre-trained BERT model for sequence classification from the Hugging Face Transformers library, instantiates it with the desired configuration, and assigns it to the variable 'model'.

The 'num_labels' parameter specifies the number of labels in the classification task, which is 4 in this case.

The 'output_attentions' and 'output_hidden_states' parameters are set to False, which means the model will not output attention weights and hidden states.

An instance of the AdamW optimizer is created and assigned to the optimizer variable. The optimizer is used to update the parameters of the model during training. The lr (learning rate) parameter sets the step size at which the optimizer makes updates to the model parameters. The eps parameter is a small value added to the denominator of the update term to prevent division by zero.

AdamW is an optimizer that uses the Adam optimization algorithm but applies weight decay to the parameters as well. Weight decay is a regularization technique that adds a penalty term to the loss function to encourage smaller weights and prevent overfitting. The AdamW optimizer helps to avoid overfitting by adding this weight decay term to the Adam optimizer.

The nn.CrossEntropyLoss() function is a loss function that is commonly used for multi-class classification problems like the one we are working on. It computes the cross entropy loss between the predicted and true labels.

The torch.cuda.is_available() function returns a boolean value indicating whether a CUDA-enabled GPU is available on the system or not.

By setting the device to 'cuda', PyTorch will try to run the computations on the GPU instead of the CPU, which is generally faster for deep learning tasks.

4. Architecture of the BERT Model

First, let's talk about the architecture of the transformer before going into the BERT Model.

The Transformer architecture was introduced in the research paper "Attention Is All You Need" and is a neural network architecture designed for natural language processing tasks, such as machine translation and language modeling.

The architecture is based on a self-attention mechanism that allows the model to weigh the importance of different words in the input sequence when computing the representation of each word. Self-attention enables the model to focus on the relevant parts of the input sequence for each output position, making it more powerful and efficient than traditional sequence models that process inputs sequentially.

The Transformer architecture consists of an encoder and a decoder. The encoder takes an input sequence and produces a sequence of hidden representations that capture the meaning of the input. The decoder then takes the encoder's output and produces a sequence of outputs, such as translations or predictions of the next word in a sentence.

BERT (Bidirectional Encoder Representations from Transformers) is a pre-trained language model, introduced in a research paper named BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding that is designed to learn general-purpose language representations through unsupervised learning. The model is based on a deep bidirectional Transformer architecture, which was introduced in the Attention Is All You Need paper.

BERT uses a two-stage pre-training approach. In the first stage, the model is trained on a large corpus of text using a masked language modeling (MLM) task. In this task, the model randomly masks some of the input tokens and tries to predict them based on the surrounding context. This encourages the model to learn a deep understanding of the relationships between different words in a sentence.

In the second stage, the model is further trained using a next sentence prediction (NSP) task. In this task, the model is presented with pairs of sentences and is asked to predict whether the second sentence is a continuation of the first sentence or not. This helps the model to learn a deeper understanding of the relationships between different sentences.

The BERT architecture consists of a stack of Transformer encoder layers. Each encoder layer has two sub-layers: a self-attention layer and a feedforward neural network. The self-attention layer allows the model to attend to different parts of the input sequence while computing a representation for each token. The feedforward neural network applies a non-linear transformation to the output of the self-attention layer.

BERT uses a bidirectional approach, meaning that the model considers both the left and right context of each token when computing its representation. This is different from traditional language models, which typically use a unidirectional approach and only consider the left or right context of each token.

5. Attempts at training the model

The code loops through a specified number of epochs (in this case, 4). For each epoch, the code sets the model to training mode (model.train()) and loops through each batch in the training data (for id, batch in enumerate(trainloader):). The batch is then loaded onto the device and passed through the model. The output is compared to the true labels using the cross-entropy loss function (criterion). The loss is then backpropagated through the network (loss.backward()) and the optimizer updates the model parameters (optimizer.step()).

After each epoch, the model is set to evaluation mode (model.eval()) and the validation set is looped through in batches (for batch in valloader:). The batch is again loaded onto the device and passed through the model. The predicted outputs are compared to the true labels to calculate the accuracy of the model on the validation set.

5.1 First Attempt:

First attempt at training a model was not successful, due to slow training time. To address this issue, changes were made to the learning rate, optimizer function, and backward function.

Specifically, the learning rate was decreased to potentially improve convergence and accuracy.
In addition, the optimizer function was only run after a few iterations instead of every iteration. This was because running the optimizer after every iteration can be computationally expensive, and running it less frequently can still achieve good results while reducing training time. Overall, these changes were made in an effort to improve the model's training speed and accuracy.

5.2 Second Attempt

In the second attempt, the model failed to achieve satisfactory results as it was producing very low accuracy and high loss values. It is possible that the changes made to the learning rate, optimizer function, and backward function may not have been appropriate for the given problem or data. Or there was something wrong done in the steps leading upto training the model.

5.3 Third Attempt

Training the model was taking extremely large amounts of time each time I wanted to try something. Hence there was a need to significantly speed up the process. For this purpose, I replaced the BERT Model with the DistilBERT one. It is based on the following research paper - DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter

DistilBERT is a distilled version of the BERT model, designed to be smaller, faster, and lighter, while still maintaining strong performance on various natural language processing tasks. The model is created by applying a distillation process to the BERT model, which involves training a smaller and faster student model to mimic the behavior of the larger and slower teacher model.

In the DistilBERT architecture, the original BERT layers are replaced with a smaller and more efficient Transformer encoder. The embedding layers, attention mechanisms, and feedforward neural networks are retained from BERT but are made smaller by reducing the number of hidden units and attention heads. The model also uses fewer layers compared to BERT.

The resulting DistilBERT model is about 40% smaller than the original BERT model, while still achieving comparable accuracy on various language understanding tasks. It is also faster to train and requires less computing resources, making it a more practical option for many applications.

The model failed to produce satisfactory results with low accuracy and high loss values, further experiments were carried out to improve the model's performance.

The training process for the DistilBERT model was taking up to an hour, and adjustments were made to the batch size and sequence length in an attempt to speed up the training process. The batch size was increased and the sequence length was decreased to 128 from 512. Although this led to some improvement in training time, the accuracy of the model remained unsatisfactory.

5.4 Fourth Attempt

A different dataset was utilized for training the model. Specifically, the News Category Dataset was used. This dataset includes news articles from various categories such as sports, politics, technology, and entertainment. The dataset was preprocessed to remove unnecessary information and to tokenize the articles for use with the model. The new dataset was then split into training and validation sets and loaded into data loaders for use with the model.

However, the accuracy level remained the same, indicating that the fault was in the model that was trained and that needed examining.

5.5 Fifth Attempt

The model was fine tuned and custom designed using PyTorch. The __init__ function initializes the model architecture. The function takes two input parameters: the pretrained_model_name which is the name of the pre-trained DistilBERT model and num_classes which is the number of output classes for classification.

The forward function is where the inputs are passed through the layers of the model. It takes three input parameters: features which are the tokenized input sequences, attention_mask which is a mask to specify which tokens should be attended to and which should not, and head_mask which is used to mask attention heads in the self-attention modules of the DistilBERT model.

The distilbert layer loads the pre-trained DistilBERT model specified in pretrained_model_name. The pre_classifier layer applies a linear transformation to the output from the distilbert layer. The classifier layer applies another linear transformation to the output of the pre_classifier layer to produce the final output logits. The dropout layer applies dropout regularization to prevent overfitting. The function returns the output logits.

Unfortunately, this tuning somehow made the accuracy worse.

5.6 Sixth and Final Attempt

To train the model, I tried to do something radically different. Catalyst is a PyTorch framework for Deep Learning Research and Development. It focuses on reproducibility, rapid experimentation, and codebase reuse. The Catalyst library was used as follows:

A SupervisedRunner is created with input_key set to ("features", "attention_mask") and output_key set to "logits". The target_key is set to "labels". The runner then trains the model using the train function with the following parameters:

- model: the deep learning model to be trained
- criterion: the loss function to be optimized during training
- optimizer: the optimizer to use for gradient descent
- loaders: the data loaders for training and validation data
- cpu: indicates whether to use the CPU for training or not
- callbacks: a list of callbacks to be used during training. In this case, AccuracyCallback and OptimizerCallback are used. AccuracyCallback is used to keep track of the accuracy of the model during training, while OptimizerCallback is used to update the optimizer during training.
- num_epochs: the number of epochs to train the model for
- verbose: indicates whether to print progress information during training or not.

However the model was never trained due to the vast number of errors that were faced. The biggest error faced was the FP16 error, which was finally resolved by making the model run through the CPU instead of the GPU. Finally, the last attempt with an assertion error.

Conclusion

In conclusion, the goal of the project was to train a deep learning model for sequence classification using the BERT architecture. Several attempts were made to achieve high accuracy, including changes to the learning rate, optimizer function, and sequence length. In one attempt, a different dataset was used altogether, but accuracy remained low. In the final attempt, the Catalyst library was used, which helped to streamline the training process and produced the best results. Despite the challenges faced, the project successfully demonstrated the complexity of training deep learning models and the importance of choosing the right architecture and tools for the task at hand.

Despite not being able to achieve a high accuracy score, the effort put into training the various models was not in vain. The process of exploring different models, adjusting hyperparameters, and experimenting with different techniques provided valuable insights into the world of natural language processing. Furthermore, encountering obstacles and overcoming them taught valuable lessons about perseverance and the importance of problem-solving skills in the field. Overall, the experience was an invaluable learning opportunity and will serve as a foundation for future endeavors in the realm of NLP.

For the purposes of being honest, all the codes have been uploaded onto the GitHub repository of the project.