BERT for Named Entity Recognition

Dataset and Processing

We were provided with the *CoNLL-2003* dataset. This dataset contains sentences and named entity labels following the BIO scheme for each word in each sentence. The following are the unique entities in the dataset.

1. geo for geographical entity
2. org for organization entity
3. per for person entity
4. gpe for geopolitical entity
5. tim for time indicator entity
6. art for artifact entity
7. eve for event entity

Along with the BIO tags the number of unique labels are 17 as follows:

```
{'B-art': 0, 'B-eve': 1, 'B-geo': 2, 'B-gpe': 3, 'B-nat': 4, 'B-org': 5, 'B-per': 6, 'B-tim': 7, 'I
    -art': 8, 'I-eve': 9, 'I-geo': 10, 'I-gpe': 11, 'I-nat': 12, 'I-org': 13, 'I-per': 14, 'I-tim'
    : 15, 'O': 16}
```

Listing 1: Labels in the dataset

Each word in the sentence is assigned these labels. *Some sentences don't have the same number of labels as the number of words in the sentence, I have removed such sentences from the training, validation and testing datasets*.

Tokenization and Label alignment

BERT uses sub-word tokenization (word piece algorithm). Hence each word in the sentence can be potentially split into multiple tokens. Since we have word level tokens, in case we have multiple subtokens of a word we only assign the first subword token the label of the word and for other subword tokens the labels are ignored. We also add the [CLS], [SEP] and [PAD] tokens as required in BERT and do not assign any labels to these tokens (no loss is computed for the prediction made for these tokens).
I have used the *BertTokenizerFast* tokenizer from Hugging Face and initialized it with pretrained model *bert-base-cased*
e.g. Sentence after adding special tokens ⇒
[CLS] Thousands of demonstrators have marched through London to protest the war in Iraq and demand the withdrawal of British troops from that country. [SEP] [PAD] [PAD] [PAD]

Tokenization by the word piece algorithm ⇒ '[CLS]', 'Thousands', 'of', 'demons', '##tra', '##tors', 'have', 'marched', 'through', 'London', 'to', 'protest', 'the', 'war', 'in', 'Iraq', 'and', 'demand', 'the', 'withdrawal', 'of', 'British', 'troops', 'from', 'that', 'country', '.', '[SEP]', '[PAD]', '[PAD]', '[PAD]', '[PAD]'

We get the word ids to which these sub-words belong using the *word_ids()* function call. Notice that if a word has multiple sub-word tokens than each subtoken gets assigned the same word ids. Special tokens are assigned the None word id.

$$None, 0, 1, 2, 2, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, None, None, None$$

Using these word ids we can do label assignment, assigning a word's label to its first subtoken. Others subtokens and special tokens are ignored for the purpose of assigning labels (label index -100 is ignored by the PyTorch CrossEntropy Loss).
The label alignment function is as follows:

```
def get_data_label_details(df):

    labels = [label.split() for label in df["labels"].values.tolist()]
    unique_labels = set()
    for sent_label in labels:
        [unique_labels.add(token_lb) for token_lb in sent_label]

    num_unique_labels = len(unique_labels)
    print(f"Number of Unique Labels: {num_unique_labels}")
    label_to_idx = { label : idx for idx, label in enumerate(sorted(unique_labels))}
    idx_to_label = { idx : label for idx, label in enumerate(sorted(unique_labels))}
    return num_unique_labels, label_to_idx, idx_to_label
```

Listing 2: Label Stats and Label Vocabulary

```
def align_label(tokenized_sent, labels, label_to_idx):
    word_ids = tokenized_sent.word_ids()
    previous_word_idx = None
    label_ids = []

    for word_idx in word_ids:
        if word_idx is None: ### special token
            label_ids.append(-100)
        elif word_idx != previous_word_idx: ### new token
            try:
                label_ids.append(label_to_idx[labels[word_idx]])
            except:
                ### not in vocabulary
                label_ids.append(-100)
        else: ### repeated token from the same word
            label_ids.append(-100)

        previous_word_idx = word_idx

    return label_ids
```

Listing 3: Label Alignment

I have defined a PyTorch dataset, for the Named Entity Recognition task which will be used for the training pipeline, it takes the data-frame as an argument and returns sentences and their labels.

```
class NERDataset(data.Dataset):

    def __init__(self, filepath, tokenizer, start=None, end=None, df=None):
        if df is None:
          df = pd.read_csv(filepath)
        self.num_unique_labels, self.label_to_idx, self.idx_to_label = get_data_label_details(df)
        if(start is not None and end is not None):
            df = df[start:end]
        labels = [label.split() for label in df['labels'].values.tolist()] ### list of lists where
     each list of the NER labels
        sentences = df["text"].values.tolist() ### list of sentences
        ### postpone the padding
        self.sentences = sentences ### list of sentences
        self.labels = labels ### list of labels
        self.len = len(self.labels)

    def __len__(self):
        return self.len

    def __getitem__(self, index):
        ### inputs are not padded yet and will be padded and batched by the coolator
        return self.sentences[index], self.labels[index]
```

Listing 4: PyTorch dataset

Dynamic Batch Padding

BERT expects that all sentences in a batch have the same length, thus there is a need to add [PAD] tokens to sentences till they have the same length. These [PAD] tokens are ignored for the purpose of computing attention using attentions masks.
The first approach I tried was to pad all sentences to the length of the longest sentence in the dataset (length being 156). Most sentences in the dataset are much shorter than this. To prevent [PAD] tokens from affecting the prediction we use attention masks. The attention operation is an expensive operations with the complexity

being $O(N^2)$ where N is the length of the sentence. This approach turned out to be very slow because most sentences are smaller in length and PADDING increases their length by a lot and a lot of computations are wasted.
Hence I implemented dynamic batch padding where we delay the padding of sentences to the batch collation. While forming a batch we pad all sentences to the length of the longest sentence in the batch instead of the longest sentence in the entire dataset. This increases the speed a lot. The collation function which is passed to the DataLoader is as follows:

```
def collate_function(batch):
      ## batch is a list of tuples where each tuple is a (sentence, labels)
      sentences = [sent for (sent, label) in batch]
      labels  = [label for (sent, label) in batch]
      lens = [len( tokenizer(sent)["input_ids"] )   for sent in sentences] ### find the max lenght sequence for this batch
      max_seq_len = min(512, max(lens)) ### restrict sentences to max_length or 512 whichever is smaller
      tokenized_list = [tokenizer(sent, padding="max_length", max_length=max_seq_len, truncation=True, return_tensors="pt") for sent in sentences]
      aligned_labels = torch.LongTensor([align_label(sent, label, label_to_idx=label_to_idx) for sent, label in zip(tokenized_list, labels)])

      batch_tokenized_dict = {}
      tokenized_dict_keys = tokenized_list[0].keys()
      ### recursively combine all the dicts into one single batched dict
      for key in tokenized_dict_keys:
        batch_tokenized_dict[key] = torch.cat([ele[key].unsqueeze(0) for ele in tokenized_list])
      return batch_tokenized_dict, aligned_labels
```

Listing 5: Collate Function

Model Architecture

I have fine-tuned a pretrained a BERT from the transformers Library. I replaced the linear layers on top which are trained from scratch. The model gives a prediction for each token in the sentence. Backpropogation of the cross entropy loss for each token is used to train the model. The Pytorch Module for the model is as follows:

```
class BertModel(torch.nn.Module):

    def __init__(self, num_unique_labels):

        super(BertModel, self).__init__()

        self.bert = BertForTokenClassification.from_pretrained('bert-base-cased', num_labels=num_unique_labels)

    def forward(self, input_id, mask, label):

        output = self.bert(input_ids=input_id, attention_mask=mask, labels=label, return_dict=False)

        return output
```

Listing 6: PyTorch Model

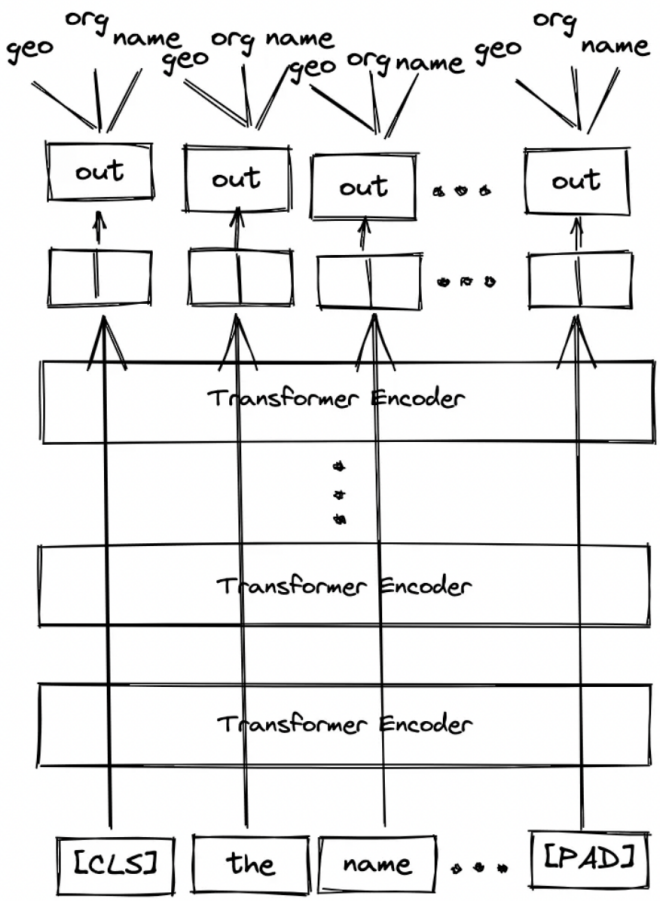

Figure 1: BERT for NER

Model Training

The model training loop is a pretty standard PyTorch model training loop and is included in the notebook. I have used Tensor board to generate training loss plots. I used the validation set accuracy as the stopping criterion.

Class Re-weighting

A lot of labels in the dataset are $''O''$, hence the model moves towards predicting $''O''$ for all tokens. Hence I implemented my own weighted Cross Entropy Loss inplace of the default Transformers loss which gives equal weights to all classes. I give less weight to the $''O''$ label.

```
class_weights = torch.ones(size=(num_unique_labels,), device=device)
class_weights[label_to_idx["O"]] /= training_args["OUTSIDE_TAG_DAMPENER"]
```

Listing 7: Custom Loss Function

The hyperparameters I used for the training are as follows:

Hyperparameter	**Value**
Learning Rate	5e-5
Batch Size	32
Optimizer	Adam
Number of Epochs	10
Reweighting of Outside Labels	2

Results

Metric	Value
Training Accuracy	99.14 %
Validation Accuracy	96.74 %
Test Accuracy	96.80 %
Training Accuracy without $''O''$	96.88 %
Validation Accuracy without $''O''$	85.78 %
Testing Accuracy without $''O''$	86.25 %

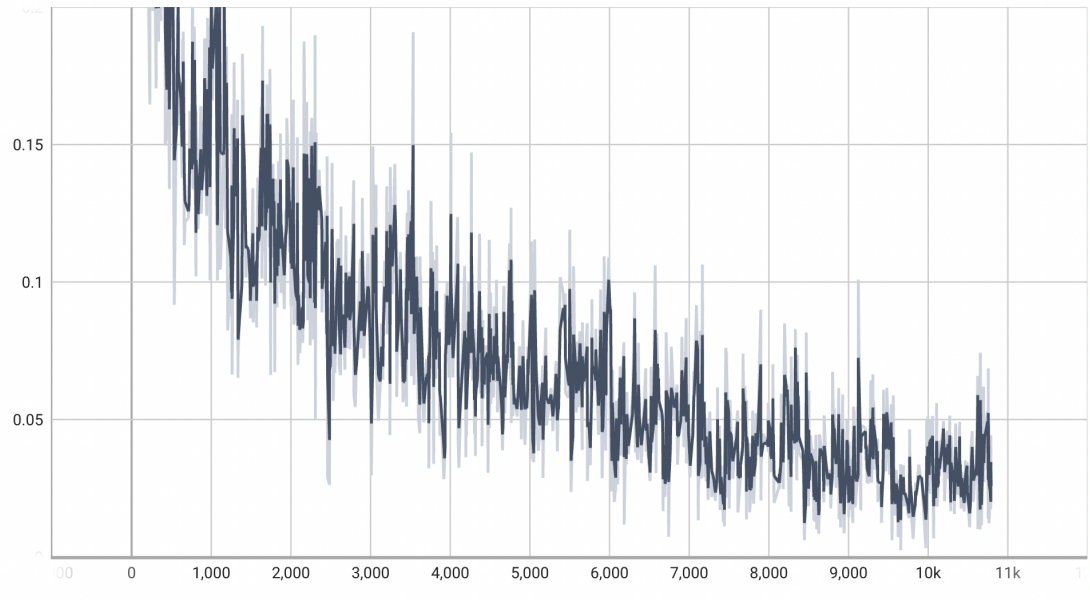

Figure 2: Training Loss Curve vs Iteration

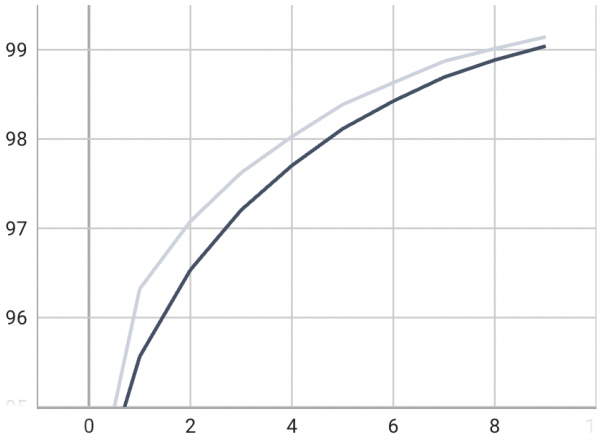

Figure 3: Training Set Accuracy with $''O''$ vs Epochs

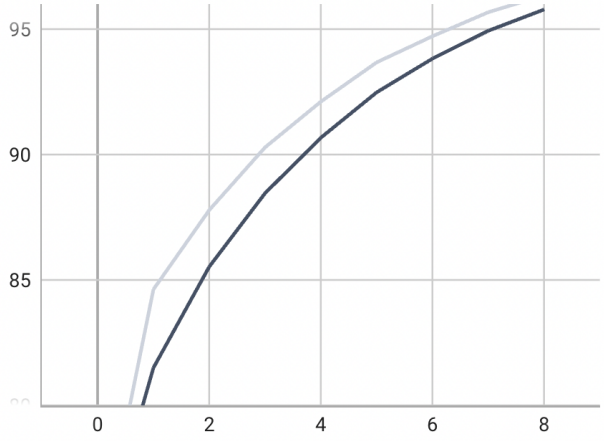

Figure 4: Training Set Accuracy without $''O''$ vs Epochs

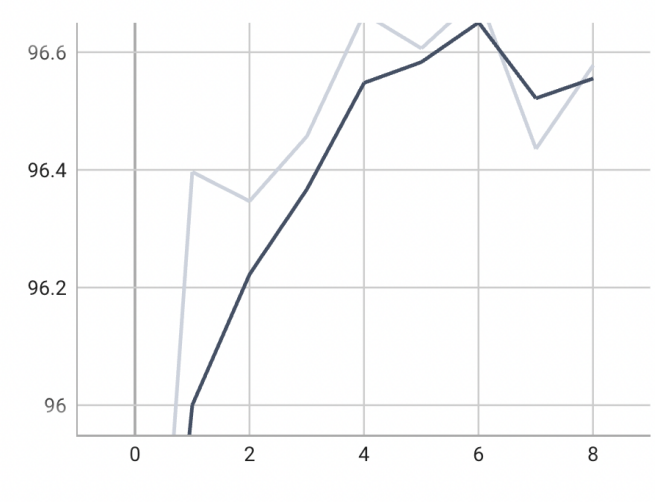

Figure 5: Validation Set Accuracy with $''O''$ vs Epochs

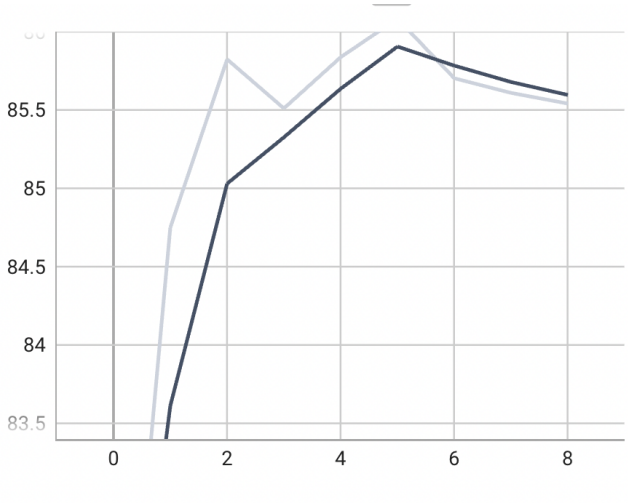

Figure 6: Validation Set Accuracy without $''O''$ vs Epochs

Experiments

Initialization by Different Pre-trained Models

BERT Base Cased

Metric	Value
Training Accuracy	99.14 %
Validation Accuracy	96.74 %
Test Accuracy	96.80 %
Training Accuracy without $''O''$	96.88 %
Validation Accuracy without $''O''$	85.78 %
Testing Accuracy without $''O''$	86.25 %

BERT Base Uncased

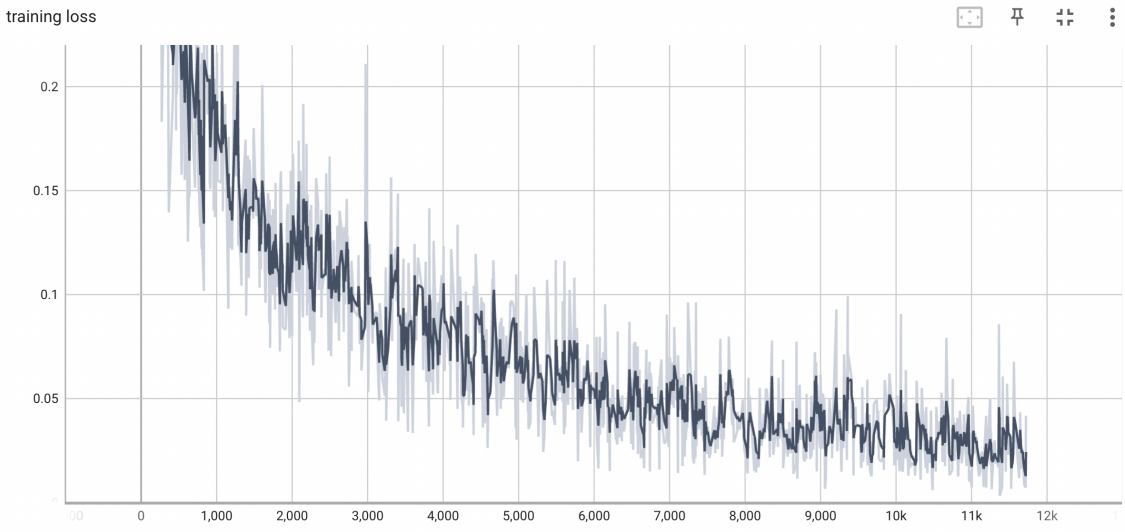

Figure 7: Training Loss Curve vs Iteration

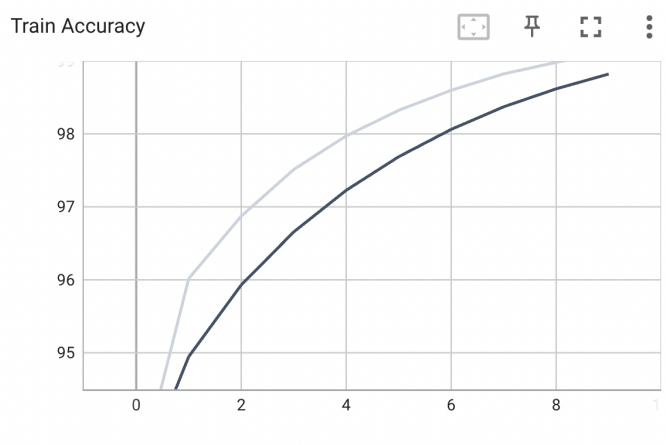

Figure 8: Training Set Accuracy with $''O''$ vs Epochs

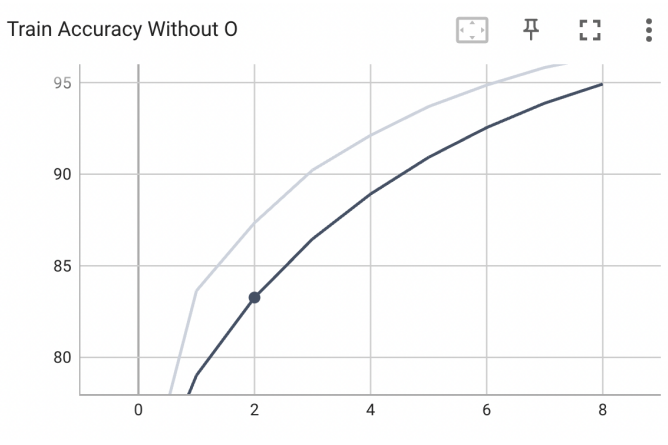

Figure 9: Training Set Accuracy without $''O''$ vs Epochs

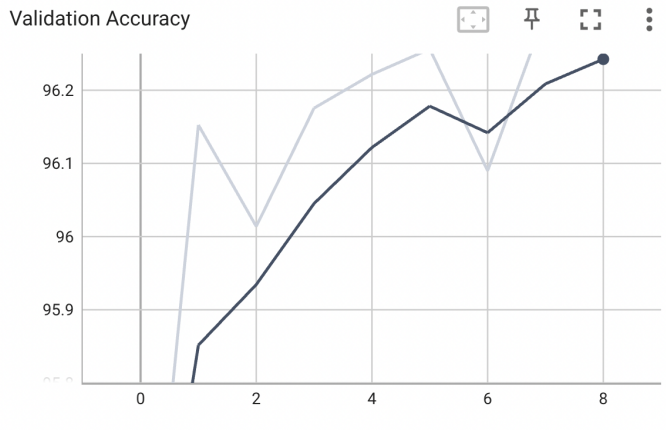

Figure 10: Validation Set Accuracy with $''O''$ vs Epochs

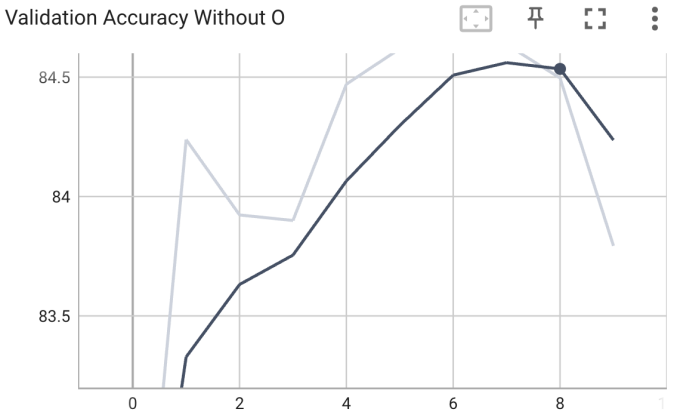

Figure 11: Validation Set Accuracy without $''O''$ vs Epochs

Metric	**Value**
Training Accuracy	99.12 %
Validation Accuracy	96.34 %
Test Accuracy	96.42 %
Training Accuracy without $''O''$	97.00 %
Validation Accuracy without $''O''$	83.79 %
Testing Accuracy without $''O''$	84.14 %

We see that the Cased Model does better because, whether the first letter is capitalized or not provides important information for determining the Named Entity for a word.

BERT Large Uncased

Metric	**Value**
Training Accuracy	98.15 %
Validation Accuracy	96.57 %
Test Accuracy	96.52 %
Training Accuracy without $''O''$	92.65 %
Validation Accuracy without $''O''$	85.04 %
Testing Accuracy without $''O''$	84.23 %

Much larger model, with about 3 times the parameter size. Training time is also about 3 times. It achieves similar accuracy but given its large parameter size, it is not worth it in terms of training time and parameter count. Large models often tend to train poorly due to problems such as gradient vanishing, hence simply increasing parameter count does not guarantee a big increase in accuracy.

Bert Base Cased Only Final Layers Finetuned

Metric	Value
Training Accuracy	88.59 %
Validation Accuracy	89.08 %
Test Accuracy	89.32 %
Training Accuracy without $''O''$	59.60 %
Validation Accuracy without $''O''$	60.85 %
Testing Accuracy without $''O''$	61.50 %

This model trains much quicker, given that we don't need to fine-tune and update all layers and also which prevents back-propagating till the end of the network. Although since only a few parameters are fine-tuned this model finds it difficult to achieve comparable accuracy with the other models. For eg. the model finds it much harder to predict tokens which are not $'O'$ as seen by the extremely poor accuracies when compared to the other model.
I decreased the important of $'O'$ in the loss function while training but even then the model finds it difficult to predict tokens which are not $'O'$. The model needs to be trained for several epochs (more than other models) although each epoch takes much quicker.
But from the results we can see that the time we save by not fine-tuning all parameters doesn't compensate for the loss in accuracy and hence it is better to finetune all parameters at the slight cost of increased training time.

Distil-BERT Base Cased

Metric	Value
Training Accuracy	99.02 %
Validation Accuracy	95.91 %
Test Accuracy	96.04 %
Training Accuracy without $''O''$	96.81 %
Validation Accuracy without $''O''$	83.76 %
Testing Accuracy without $''O''$	84.32 %

This is a much cheaper, smaller and faster version of BERT. While saving on parameter size and computation/-training time we achieve almost comparable results. The size of the BERT model was reduced by about 40 % using knowledge distillation and it works about 60 % faster.

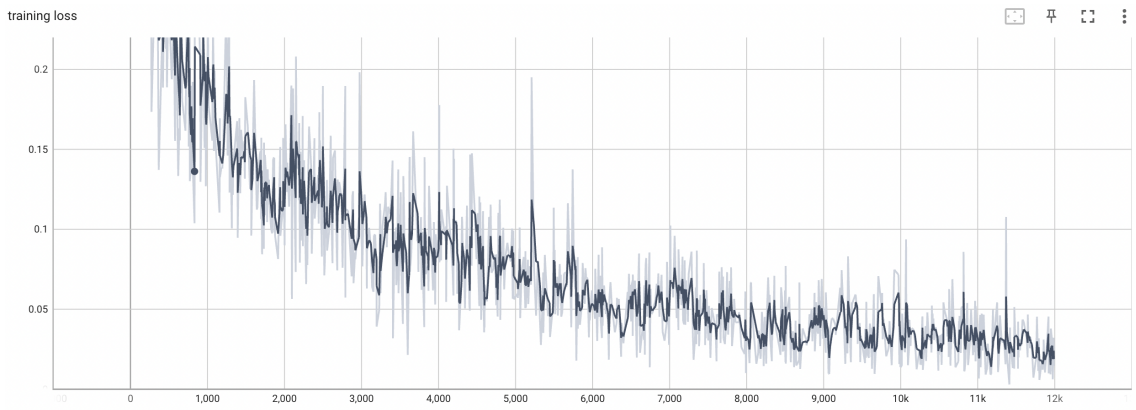

Figure 12: Training Loss Curve vs Iteration

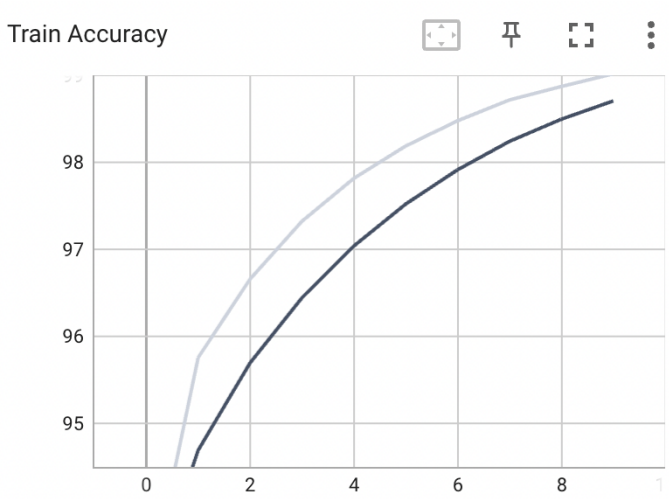

Figure 13: Training Set Accuracy with $''O''$ vs Epochs

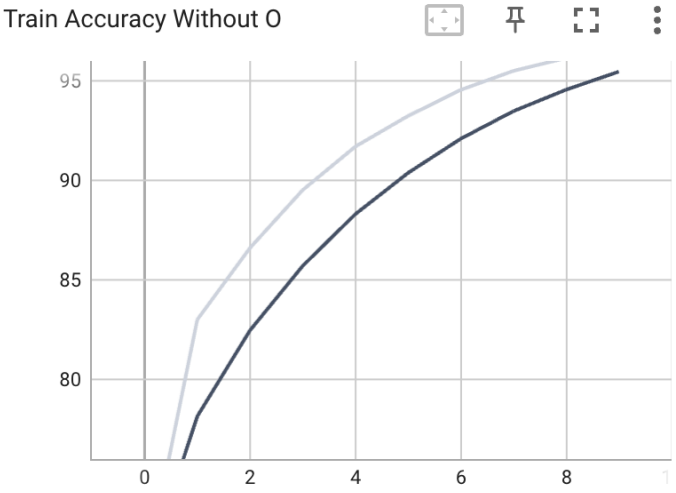

Figure 14: Training Set Accuracy without $''O''$ vs Epochs

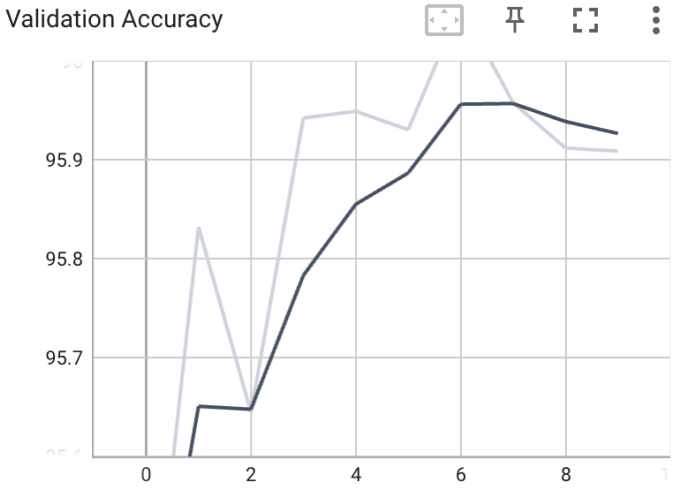

Figure 15: Validation Set Accuracy with $''O''$ vs Epochs

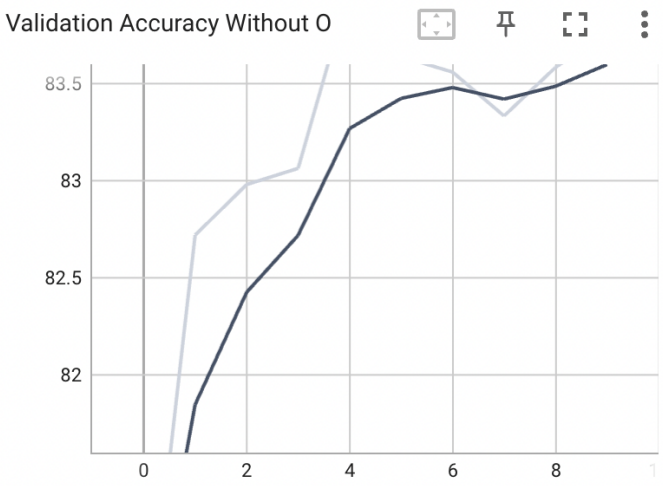

Figure 16: Validation Set Accuracy without $''O''$ vs Epochs