Strategies for Addressing Model Training Issues and Architecture Overview

AprilFour

2024

1 Introduction

This document outlines the strategies employed to resolve common model training issues such as overfitting, underfitting, and data leakage during the training of a BERT-based Named Entity Recognition (NER) model. It also provides a description of the BERT model architecture used in this project and the baseline function for computing evaluation metrics. Additionally, this document discusses adjustments made to the training arguments to optimize model performance.

2 Model Architecture: BERT for NER

The architecture of the BERT model used for Named Entity Recognition is based on the **BERT** (Bidirectional Encoder Representations from Transformers) model, which utilizes self-attention mechanisms to capture contextual relationships between words in a sentence. The architecture comprises the following key components:

- **Embedding Layer:** Converts input tokens into dense vectors of fixed size, representing both the token and its position in the sequence.
- **Transformer Encoder Layers:** Consists of multiple stacked transformer blocks, each containing self-attention and feed-forward neural networks. BERT employs bidirectional attention, allowing it to attend to tokens both before and after a given word.
- **Output Layer:** For NER tasks, a classification head is applied to each token. The model predicts the entity label (e.g., PERSON, ORGANIZATION, etc.) for every token in the input sequence.
- **Pre-trained and Fine-tuned:** The BERT model is pre-trained on a large corpus and fine-tuned on a labeled NER dataset to adapt it to the specific task.

The model is trained to minimize cross-entropy loss over the token classifications, using a labeled dataset for supervised learning.

3 Baseline Function for Metrics

The function below provides the baseline for computing key evaluation metrics such as accuracy, precision, recall, and F1-score. These metrics are crucial for assessing the model's performance during training and validation.

Listing 1: Baseline Function for Metrics

```
def compute_metrics(p):
    preds = p.predictions.argmax(-1)
    labels = p.label_ids
    precision, recall, f1, _ =
    precision_recall_fscore_support(labels, preds, average='weighted')
    acc = accuracy_score(labels, preds)
    return {
        'accuracy': acc,
        'f1': f1,
        'precision': precision,
        'recall': recall
    }
```

4 Training Issue 1: Overfitting

Overfitting occurs when the model performs exceptionally well on the training data but fails to generalize to unseen data. The following steps were taken to address this:

- **Step 1: Early Stopping** - Introduced early stopping to halt training when validation loss stopped improving for a certain number of epochs.
- **Step 2: Data Augmentation** - Increased dataset diversity through data augmentation techniques, such as synonym replacement and sentence reordering.
- **Step 3: Regularization** - Added L2 regularization (weight decay) to prevent the model from learning overly complex patterns.
- **Step 4: Dropout** - Increased the dropout rate in the transformer layers to reduce model complexity and prevent overfitting.
- **Step 5: Reduced Model Size** - Experimented with smaller BERT variants (e.g., DistilBERT) to reduce the risk of overfitting.

5 Training Issue 2: Underfitting

Underfitting occurs when the model is too simplistic to capture the underlying patterns in the data, leading to poor performance on both training and validation sets. To address underfitting, the following strategies were employed:

- **Step 1: Increase Model Capacity** - Used larger BERT models (e.g., BERT-large) to increase the model's capacity to capture complex relationships.
- **Step 2: Fine-Tuning** - Fine-tuned the pre-trained BERT model for a longer duration with a smaller learning rate.
- **Step 3: More Epochs** - Increased the number of training epochs to allow the model sufficient time to learn from the data.
- **Step 4: Batch Size Tuning** - Experimented with different batch sizes to balance convergence speed and stability.
- **Step 5: Feature Engineering** - Added additional features or embeddings to enhance the model's input representation.

6 Training Issue 3: Data Leakage

Data leakage occurs when information from the validation set or test set leaks into the training set, resulting in overly optimistic performance metrics. To prevent data leakage, the following measures were implemented:

- **Step 1: Proper Data Splitting** - Ensured that the training and validation sets were split correctly using a random seed and avoided any overlap between them.
- **Step 2: Deduplication** - Removed any duplicate examples from both the training and validation sets to avoid repeated samples.
- **Step 3: Cross-Validation** - Employed cross-validation to ensure robust model evaluation by using different subsets of the data for training and validation.
- **Step 4: Stratified Sampling** - Used stratified sampling to ensure that the distribution of labels was consistent between the training and validation sets.
- **Step 5: Monitoring** - Regularly monitored metrics across both the training and validation sets to detect any signs of data leakage.

7 Training Issue 4: Repetitions in the Dataset

Repetitions in the dataset can lead to memorization and artificially inflated performance metrics. The following steps were taken to handle such issues:

- **Step 1: Remove Duplicates** - Ran a duplicate check on the dataset and removed repeated examples to avoid redundancy.
- **Step 2: Balance Dataset** - Ensured a balanced dataset with diverse samples for both positive and negative classes.
- **Step 3: Data Augmentation** - Augmented the dataset with additional, unique examples to increase the model's generalization capabilities.
- **Step 4: Data Scrutiny** - Carefully examined the dataset to avoid including redundant patterns that could mislead the model.
- **Step 5: Validation Check** - Verified that the validation set did not contain any duplicated examples from the training set.

8 Adjusting Training Arguments (Hyperparameters)

To optimize the training process and improve model performance, various training arguments (hyperparameters) were adjusted during the fine-tuning process. The key hyperparameters and their adjustments are detailed below:

- **Step 1: Learning Rate** - Experimented with different learning rates. Initially, a learning rate of 5×10^{-5} was used, but this was later reduced to 2×10^{-5} to allow for more fine-grained learning as training progressed.
- **Step 2: Batch Size** - Adjusted the batch size to improve gradient updates. A batch size of 16 was found to be effective for striking a balance between computational efficiency and model stability.
- **Step 3: Epochs** - Increased the number of epochs from 3 to 5 based on early stopping criteria, ensuring the model had enough iterations to converge without overfitting.
- **Step 4: Warmup Steps** - Set warmup steps to 10% of total steps to ensure the learning rate increased gradually, preventing large updates early in training.
- **Step 5: Weight Decay** - Applied a weight decay of 0.01 to reduce overfitting by penalizing large weights in the model.

These adjustments were made iteratively, depending on model performance after each training run. Monitoring the loss curves and evaluation metrics helped identify which hyperparameters required fine-tuning.

9 Conclusion

This document outlines the strategies employed to address various training issues during the development of the BERT NER model. Adjusting hyperparameters like learning rate, batch size, and weight decay, along with handling issues like overfitting, underfitting, data leakage, and dataset repetition, helped ensure robust model performance. By monitoring and fine-tuning these aspects, the model was successfully trained to generalize well across unseen data.