Faculty of Natural and Mathematical Sciences
Department of Informatics

King's College London,
Strand Campus, London,
United Kingdom

7CCSMPRJ
Individual Project Submission 2021/22

Name: Chia-Jung, Chang
Student Number: 21020221
Degree Programme: MSc Artificial Intelligence
Project Title: A Multi-Linear Relationship Transformer Framework for Multi-Task Sequence Labelling
Supervisor: Helen, Yannakoudakis
Word count: 11,012

Signature:

CHIA JUNG CHANG

Date: 02-08-2022

Department of Informatics
King’s College London
United Kingdom

7CCSMPRJ MSc Project

A Multi-Linear Relationship Transformer Framework for Multi-Task Sequence Labelling

Name: Chia-Jung, Chang
Student Number: 21020221
Degree Programme: MSc Artificial Intelligence

Supervisor’s Name: Helen, Yannakoudakis

This dissertation is submitted for the degree of MSc in Artificial Intelligence

ABSTRACT

There has been a tremendous amount of work and progress on sequence labelling; meanwhile, Multi-Task Learning methods are widely used for training neural networks to extract meaningful information among tasks, including the Transformer-based Multi-Task models. Whereas, the performance of the models may drop when there is no accurate representation of the relationships between different tasks. Therefore, this project mainly aims to propose a Multi-Linear Relationship Transformer framework with tensor alignment to mitigate such issues for Multi-Task sequence labelling. To achieve this, the framework will be evaluated to see if it can obtain better results and indeed estimate the relations among tasks, features, and labels. Besides, since different Transformer encoders (BERT, BERT-BiLSTM, and RoBERTa) may lead to various outcomes, the encoders will be compared to see which one is greater for Multi-Task sequence labelling. The contribution of the project is mainly that the connection between the Multi-Linear Relationship learning approach and Multi-Task sequence labelling is built to make the Transformer-based models perform better. The outcomes of the experiment on the CoNLL-2003 and the CoNLL-2000 datasets show that the framework can generally regularize the Multi-Task models to produce better results, and the quality of the task, label and feature relations learnt by the models can be visualized properly to illustrate that the framework can estimate the relations. Apart from these, it is also found that the framework can perform well even if there are few samples. As the results indicate, it is hard to conclude which Transformer encoder is better for Multi-Task sequence labelling, but the proposed framework outperforms its Multi-Task Learning counterpart on most occasions.

ORIGINALITY AVOWAL

Chia-Jung, Chang
August 2, 2022

ACKNOWLEDGEMENT

Besides my efforts, the project is completed with the assistance of my individual project supervisor Dr Helen Yannakoudakis. With the help and feedback from her, I can revise and improve my work efficiently. Therefore, I would like to acknowledge her for her guidance and suggestion throughout the project.

NOMENCLATURE

$\mathbf{b_i}$, $\mathbf{b_c}$, $\mathbf{b_o}$, $\mathbf{b_f}$	The bias parameters for hidden layers in an LSTM unit
$\mathbf{\hat{c}_t}$	The temporary state information of an LSTM cell
$\mathbf{c_t}$	The updated state information of an LSTM unit
d, D, k, K	The size of a dimension
$\mathbf{f_t}$	The result from the forget gate in an LSTM unit
$\mathbf{h_t}$	The output from a complete LSTM unit
$\mathbf{h_t^b}$, $\mathbf{h_t^f}$	The Hidden states of $\mathbf{LSTM^b}$ and $\mathbf{LSTM^f}$, respectively
$\mathbf{i_t}$	The information for remembering in an LSTM cell
l	A task in L
o_s^l	The output label sequence corresponding to s in task l
$\mathbf{o_t}$	The result from the output gate in an LSTM unit
$p(\cdot)$	A probability density function
s	A sequence in S output label sequences
$\mathrm{softmax}(\cdot)$	A softmax transfer function
t, t_i	A task in T
$\tanh(\cdot)$	A hyperbolic tangent activation function
$\mathbf{u_t}$	The combined information of LSTMs
$\mathrm{vectorize}(\cdot)$	A function for flattening a tensor into a vector
$\mathbf{w}_i$	A parameter matrix in $\boldsymbol{W}$
$\mathbf{x_t}$	The representation for current input in an LSTM cell
$\mathbf{x} \sim N(\boldsymbol{\mu}, \boldsymbol{\Omega})$	The vector $\mathbf{x}$ follows a multivariate normal distribution with the mean vector $\boldsymbol{\mu}$ and the covariance matrix $\boldsymbol{\Omega}$
$(\mathbf{x}_j^{t_i}, \mathbf{y}_j^{t_i})$	A training sample pair of the task t_i, where $\mathbf{x}_j^{t_i}$ is the input vector, and $\mathbf{y}_j^{t_i}$ is a classification label for $\mathbf{x}_j^{t_i}$
$\mathbf{y_t}$	The final output label distribution of a BiLSTM model
C	The total number of all classes (labels) across all tasks
$\mathbf{C_{diag}}$	The diagonal matrix of $\mathbf{C}$
$\mathbf{D_i}$, $\mathbf{D_f}$, $\mathbf{D_c}$, $\mathbf{D_o}$	The parameters of an LSTM cell for the input vector
E_i	The sum of (segmentation, token, and position) embeddings generated by a BERT tokenizer in advance corresponding to $Tok\ i$
$F_i(\mathbf{x}_j^{t_i})$	The estimation of $\mathbf{y}_j^{t_i}$ produced by the composite model
$\mathbf{I}_D$	An identity matrix with width and height D
L	A set or a list of sequence labelling tasks
L_t	The cross entropy loss for a sequence labelling task
$\mathrm{L}(\cdot)$	A loss function
L_{reg}	The regularization term under MRN
$\mathbf{LSTM^b}$	A backward LSTM
$\mathbf{LSTM^f}$	A forward LSTM
$\boldsymbol{M}$	A mean tensor
O	A set of output label sequences
$\boldsymbol{O}$	A zero tensor
S	A set of sequences
T	A task set under MTL or MRN
$\mathbf{T}_i$	The hidden output of BERT corresponding to $Tok\ i$
$Tok\ i$	The original i-th input token

$\boldsymbol{W}$	A parameter tensor
$\boldsymbol{W}_{..j}$	The [:, :, j] slice gained from $\boldsymbol{W}$
$\mathbf{W}_{(k)_i}$	The row with the index i from $\mathbf{W}_{(k)}$
$\mathbf{W_b}$, $\mathbf{W_f}$, $\mathbf{W_y}$	The weight matrices of the specified hidden states of an LSTM unit
$\mathbf{W_i}$, $\mathbf{W_f}$, $\mathbf{W_c}$, $\mathbf{W_o}$	The parameters for hidden layers of an LSTM unit
$\mathbf{X}$	A matrix
$\mathbf{X}_{(i)}$	The matrix gained from the mode-i flattening to the tensor $\boldsymbol{\chi}$
$\sigma(\cdot)$	A sigmoid function calculating element-wisely
$\odot$	A product operator calculating element-wisely
$\otimes$	A matrix direct product operator
$\mathbf{\Omega}$, $\mathbf{C}$	A covariance matrix
ε	A relatively small value to make an equation more stable
$\boldsymbol{\chi}$	A tensor or the tensor containing all input samples
$\boldsymbol{\chi} \sim TN(\boldsymbol{M}, \mathbf{\Omega}_1, \mathbf{\Omega}_2, \dots, \mathbf{\Omega}_k)$	The tensor $\boldsymbol{\chi}$ follows a tensor-variate normal distribution with the mean tensor $\boldsymbol{M}$ and the covariance matrices $\mathbf{\Omega}_1$ to $\mathbf{\Omega}_k$
$\{\boldsymbol{\chi}_{train}^{l_i}, \boldsymbol{\Upsilon}_{train}^{l_i}\}$	The training set for $l_i \in L$
$\{\boldsymbol{\chi}_{test}^{l_i}, \boldsymbol{\Upsilon}_{test}^{l_i}\}$	The testing set for $l_i \in L$
$\boldsymbol{\Upsilon}$	The tensor containing all classification labels

CONTENTS

List of Tables

LIST OF FIGURES

1 INTRODUCTION

Sequence labelling is a kind of classic task in the field of Natural Language Processing (NLP), whose goal is assigning labels to tokenized text so that each token has a label. Text chunking, Part-of-speech (POS) tagging and Named-entity Recognition (NER) [1] are well-known sequence labelling tasks and have been continuously studied, and they are also important approaches to extracting information from target text. Furthermore, some Multi-Task Learning (MTL) [2] methods have been broadly used and extended for NLP tasks, where sequence labelling is included. Instead of training models with only one task, MTL mainly seeks to progress certain abilities of models to generalize well for multiple tasks.

When there are multiple sequence labelling tasks, a Single-Task Learning (STL) model may fail to simultaneously cope with more than one task and acquire poor results. Consequently, as there are multiple tasks, various STL models are needed for each task instead of only one single model. For example, a POS tagging model can fail a NER task since the model is only trained with POS tagging data under STL. Hence, a POS model and a NER model are individually trained for the demand of overall performance, while MTL models can efficiently learn tasks together. Furthermore, except for the comparison mentioned above, in a study [3], it is pointed out that MTL models may perform better than STL models because MTL can share useful information between tasks, which is another advantage of MTL. If tasks are related or useful to one another, MTL models may leverage the knowledge among the tasks.

However, there are challenges for sequence labelling MTL models, and they will be addressed together with the motivations, aims, objectives and contributions of this project in Section 1.1. Besides, Section 1.2 will also provide a review of the literature. In the project, a modified symmetric [24] Multi-Linear Relationship Network (MRN) [32] with a single shared Transformer [7] encoder layer is introduced for Multi-Task sequence labelling, and the framework will be tested to see whether it can enhance the abilities of sequence labelling models in terms of scalability, explicitly modelling task relatedness, and performance (F1-score) on several benchmark sequence labelling datasets. Also, different Transformer encoders will be compared to see which one can perform better on the datasets. The outcomes of the experiment indicate that the framework can make the models estimate explicit relations to obtain better results in comparison to its MTL counterpart, but it cannot be summarized which Transformer encoder is more powerful for Multi-Task sequence labelling.

1.1 Aims and Objectives

When it comes to the motivations of the project, they are strongly related to the advantages and drawbacks of MTL. As the major advantages mentioned earlier, compared to STL models, MTL models are generally more space-saving (parameter-sharing) and may perform better when tasks are strongly related. While there are benefits to using MTL, some problems may arise due to various reasons. First of all, the size of the model still increases with the number of tasks when the tasks do not share identical bottom layers, which leads to a burden for training. Although the final performance may be good, the training stage of the model is not resource-efficient. Secondly, some potential problems may occur without modelling task relatedness under MTL. Most recent proposed MTL models are composed of shared (bottom) layers together with task-specific (top) layers. In these models, the shared layers are usually seen as the basis and concatenated by the upper task-specific layers so that the shared layers can learn general features from input samples, and then the task-specific layers can extract information from the output features of the shared layers for corresponding tasks [15], [20], [23]. Whereas, except for [23], even if these layers are designed, some transferring problems may occur if the relationships between tasks are not explicitly modelled. In a word, when there is no explicit representation for task relatedness, the performance of the models may drop, so how to make the models accurately learn the task relations becomes important. Accordingly, since [5] claims that relationship learning frameworks like Multi-Task Relationship Learning (MTRL) can force models to learn precise task relations to improve their performance, it is natural to think that these frameworks can be applied to solving the transferring problem. Nevertheless, how to apply task relationship methods is troublesome because most of the original methods are designed only for binary classification (or numeric regression) tasks, not to mention that the base learners used in [5] are too simple. To solve the problem, MRN can be used since MRN can deal with multi-class classification tasks and explicitly model task covariance, class covariance and feature covariance. Last but not least, extended from the second problem, how to combine MRN and the recent powerful Transformer models for sequence labelling has been a challenge so far. MRN is only used for Multi-Task image classification in the original paper [32], so the way to regularize the Transformer models with MRN is still problematic.

With the addressed motivations, the project aims to propose a modified MRN framework with a shared Transformer encoder layer for sequence labelling to utilize the advantages of MTL and alleviate the above problems at the same time. In addition, because different Transformer encoders may achieve various outcomes, the project seeks to identify which one is better for Multi-Task sequence labelling. To achieve the aims, several objectives are listed below:

- Modify the original MRN to make it compatible with sequence labelling models.
- Evaluate the designed MRN framework by comparing the performance of the framework to its STL and MTL counterparts on the datasets to see its pros and cons.
- See if the MRN can estimate the relations between sequence labelling tasks.
- Compare different shared Transformer encoder layers (BERT [7], RoBERTa [18], BERT-BiLSTM [19]) and analyse their performance on the benchmark datasets.

The key contributions of this project are:

- Providing a way to apply MRN to sequence labelling.
- MRN can improve the performance (F1-score) of BERT, BERT-BiLSTM and RoBERTa on the CoNLL-2000 [36] and the CoNLL-2003 dataset [25] compared to MTL without regularization.
- MRN can improve the performance when there are few sequence labelling samples compared to pure MTL.

1.2 Literature Review

As Section 1.1 referred to, a majority of recent MTL models have shared layers with task-specific layers. The study [4] points out that sequence labelling tasks can be noisy to one another, so acquiring accurate relations of tasks is significant. Nevertheless, the relationships of tasks are not given in most cases, so placing a prior on model parameters and estimating the relations from data become reasonable. In [3], “what to share” and “how to share” among samples can be the bases to categorize various MTL approaches, and a type of such approach is the “Task Relation Learning Approach” that seeks to model task relations with the similarity among tasks, in terms of task covariance, task correlation, etc. Within this category, MTRL is introduced in [5], where a matrix-variate [28] normal prior is put on the model parameter matrix. After the invention of MTRL, another approach called Multi-Task Boosting (MTBoost) is further extended [6]. Either MTRL or MTBoost puts a matrix-variate Gaussian prior on the model parameter matrix, adding a regularization term after an empirical loss function. Also, task covariance matrices can be calculated from the model parameter matrices to describe pairwise task relations in both of the paradigms. As [5] and [6] show, the models can precisely derive the correlations between tasks from the calculated covariance matrices. However, the models used in the studies are rather simple (Support Vector Machines) (SVMs) [26], and they are designed only for binary classification and numeric regression problems. Therefore, in [32], MRN is proposed to extend task relationship methods to Deep Neural Networks (DNNs). For DNNs, the original priors used in [5] and [6] are not applicable since the order of the model parameter tensor is generally higher. Because of this, a new prior which is tensor-variate can be placed on the parameter tensor of a task-specific layer. With this prior, MRN can model the accurate relations between tasks, between classes, and between features with a task covariance matrix, a class covariance matrix, and a feature covariance matrix, respectively. Although MRN has been used for Multi-Task image classification, it has not been applied to sequence labelling tasks yet. Because of these reasons, the project seeks to extend MRN to sequence labelling tasks to exploit task relations.

Regarding tasks in NLP, before the invention of BERT (Bidirectional Encoder Representations from Transformers) [7], early popular NLP models such as Long Short Term Memory cells (LSTMs) and Gated Recurrent Units (GRUs) are used for most applications [8], [9], [10], [11], [12]. Whereas, in many NLP tasks, early models have been outperformed by BERT with the self-attention mechanism [13], [14] since it was proposed. Apart from sequence labelling, in the context of Single-Task text classification, a shared BERT encoder layer is concatenated by Convolutional Neural Networks (CNNs) or Bi-gated GRUs (Bi-GRUs) [16] to utilize information from different locations in the output from BERT layer. By contrast, as for the applications of Transformer-based STL models for sequence labelling, research [17] introduces a model based on RoBERTa [18]. In [17], RoBERTa is combined with Bi-directional LSTMs (Bi-LSTMs) and Conditional Random Fields (CRFs) [10], and the composite model is designed for a special sequence tagging task called “Chinese Grammatical Error Diagnosis (CGED)”. Furthermore, instead of RoBERTa, BERT is placed as a lower encoding layer of the composite model in [19], where the BERT-BiLSTM-CRF model is formed to detect named entities in Chinese Electronic Health Records. With the proposal of Transformer-based models, there are more and more applications for downstream tasks in NLP, including sequence labelling tasks.

Expect for composite BERT-like STL models, to utilize the advantages of MTL for NLP, some researchers introduce Transformer encoders as components in MTL models. For instance, researchers in [15] propose Multi-Task DNNs (MT-DNNs). MT-DNN contains a Transformer encoder layer placed at the bottom of the shared layers to produce contextual representations, and task-specific layers are deployed to leverage useful information for corresponding tasks. Similarly, several researchers propose Transformer-based MTL models for further applications. For instance, in [23], a shared BERT-BiLSTM layer is deployed for dealing with a special task which seeks to detect emotions and the corresponding causes from emotional texts. While the relevance of tasks is accurately estimated in [23], the authors use an asymmetric [24] model structure. Namely, it is known or assumed in advance that some tasks are beneficial to the target task. Whereas, it could be argued that the relations of tasks are not given on most occasions, so a symmetric model structure may be more efficient to estimate task relationships from data.

As for Multi-Task sequence labelling, in [20], HoogBERTa, which is designed based on RoBERTa, is used for multiple Thai sequence labelling tasks. Although HoogBERTa is not regularized under any relationship-learning framework, [20] provides an overall picture that different fine-tuning [7] approaches have various goals. For example, the Multi-Task Fine-tuning strategy stated in [20] is identical to MTL with shared feature layers, and the goal of the approach is to minimize a specified objective function of fine-tuning the whole Transformer composite model. On the contrary, Single-Task Fine-tuning is the same as STL, but for Single-Task Fine-tuning methods where tasks do not initially share the identical bottom encoder, the overall model size grows noticeably when the number of tasks increases. A similar problem happens in [21]. As [21] introduces, task-individual encoders are customized for each task, while the whole model can label sequences well under a symmetric adversarial (a generator and a discriminator) [22] MTL paradigm, the number of model parameters rises sharply with the number of sequence labelling tasks due to the large size of BERT (110M trainable parameters in $\text{BERT}_{\text{base}}$). In addition, the training process of the model is time-consuming and resource-demanding since the number of parameters is huge. Overall, having a shared Transformer-based encoder is beneficial to the scalability (not taking up too much space) of the models and resource-efficient, which is also one of the main purposes of the project.

Based on the review of literature, it could be summarized that a symmetric MRN sequence labelling model whose bottom Transformer-based encoder is shared may be promising.

1.3 Report Structure

The second chapter will give the background theories relevant to the project. Chapter 3 will provide the recall of objectives with the requirements and specifications of the project, and the design for achieving the aim will also be given. Then, the benchmark datasets will be referred to and explained in the fourth chapter, and Chapter 5 will be focused on the implementation of the methodology. Chapter 6 will show the obtained results, and the results will be analysed for evaluating the learning paradigms and the composite models. Chapter 7 will provide relevant issues which may arise in the project, and Chapter 8 will be the conclusion of the project. The ninth chapter will provide the references, and the tenth chapter will give the code for implementation.

2 Background Theories

In Chapter 2, the background theories relevant to the project will be provided. The following subsections will elaborate on important techniques, problems and models.

2.1 Multi-Task Sequence Labelling

As Section 1 introduced, a sequence labelling task is to tag every tokenized text with a label. Similarly, the goal of a Multi-Task sequence labelling problem [27] is that each tokenized text is assigned a label for each task. Namely, every token has multiple labels, and each label for the token is associated with a corresponding sequence labelling task. In this project, the definition of a Multi-Task sequence labelling problem is:

- Given a list or a set of sequence labelling tasks L;
- Given input sequences $S = \{s_1, s_2, \dots, s_n\}$, where n indicates the total number of sequences in S;
- Generating the set of output label sequences $O = \left\{o_{s_1}^l, o_{s_2}^l, \dots, o_{s_n}^l\right\}$ for each $l \in L$.

Therefore, as Figure 1 illustrates, when $L = \{l_1, l_2\}$, and for each $l \in L$, $o_{s_i}^l$ is generated from each input sequence $s_i \in S$.

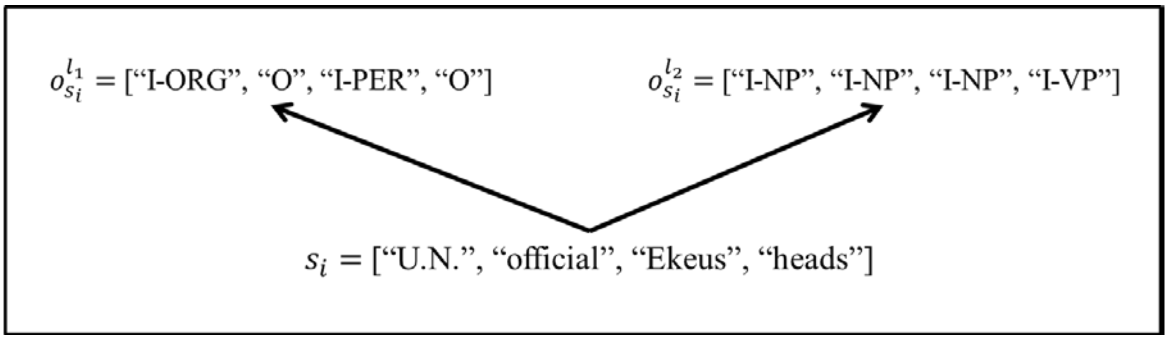

$o_{s_i}^{l_1}$ = ["I-ORG", "O", "I-PER", "O"]
$o_{s_i}^{l_2}$ = ["I-NP", "I-NP", "I-NP", "I-VP"]
s_i = ["U.N.", "official", "Ekeus", "heads"]

Figure 1: An example of the Multi-Task sequence labelling problem. Assume s_i is ["U.N.", "official", "Ekeus", "heads"]. When there are only two tasks, there should be two sequences of labels generated from s_i. Note that each token in s_i is assigned a label ("U.N." is assigned "I-ORG" in $o_{s_i}^{l_1}$, and it is labelled "I-NP" in $o_{s_i}^{l_2}$, etc.).

2.2 Tensor Normal Priors

MTRL is introduced in [5], and it aims to model the relationships between tasks. Whereas, MTRL fails to be extended for DNNs since its prior is matrix-variate. By contrast, a tensor Gaussian (normal) prior can be placed on a model parameter tensor to model task relations.

A tensor $\boldsymbol{\chi}$ with order k can be denoted as $\boldsymbol{\chi} \in \mathbb{R}^{d_1 \times d_2 \times \ldots \times d_k}$, and d_i represents the size of the i-th dimension of $\boldsymbol{\chi}$. By contrast, a matrix $\mathbf{X} \in \mathbb{R}^{d_1 \times d_2}$ is a tensor with order 2. Therefore, the formulation for $\boldsymbol{\chi}$ is an extension of representing a matrix. In [32], the tensor Gaussian distribution is represented in (1).

$$\text{vectorize}(\boldsymbol{\chi}) \sim N(\text{vectorize}(\boldsymbol{M}), \boldsymbol{\Omega}_1 \otimes \boldsymbol{\Omega}_2 \otimes \ldots \otimes \boldsymbol{\Omega}_k), \qquad (1)$$

In (1), vectorize$(\cdot)$ is a function for flattening a tensor to a vector so that vectorize$(\boldsymbol{\chi})$ has the size of $\prod_{i=1}^{k} d_i$. $\boldsymbol{M}$ denotes the mean tensor. $\boldsymbol{\Omega}_i \in \mathbb{R}^{d_i \times d_i}$ is the covariance matrix of the matrix $\mathbf{X}_{(i)}$. $\mathbf{X}_{(i)}$ can be gained by applying mode-i flattening to $\boldsymbol{\chi}$ so that each row in $\mathbf{X}_{(i)}$ has the values with the identical i-th index of $\boldsymbol{\chi}$. $\mathbf{x} \sim N(\boldsymbol{\mu}, \boldsymbol{\Omega})$ denotes that the vector $\mathbf{x}$ follows a multivariate normal distribution, where $\boldsymbol{\mu}$ indicates the mean vector and $\boldsymbol{\Omega}$ denotes the covariance matrix. In addition, in the paper [32], it is claimed that $\boldsymbol{\Omega} = \boldsymbol{\Omega}_1 \otimes \boldsymbol{\Omega}_2 \otimes \ldots \otimes \boldsymbol{\Omega}_k$, which indicates that $\boldsymbol{\Omega}$ is modelled by the matrix direct product (Kronecker product) of all the covariance matrices of each $\mathbf{X}_{(i)}$.

As [32] referred, the probability density function for (1) becomes:

$$p(\mathbf{x}) = \left(\prod_{i=1}^{k} |\boldsymbol{\Omega}_i|^{\frac{-d_{total}}{2d_i}}\right) \exp\left(\frac{\mathbf{c}^{\mathbf{T}}\boldsymbol{\Omega}^{-1}\mathbf{c}}{-2}\right) \times (2\pi)^{\frac{-d_{total}}{2}}, \qquad (2)$$

In (2), $d_{total} = \prod_{i=1}^{k} d_i$, $\boldsymbol{\Omega} = \boldsymbol{\Omega}_1 \otimes \boldsymbol{\Omega}_2 \otimes \ldots \otimes \boldsymbol{\Omega}_k$, $\mathbf{c} = \text{vectorize}(\boldsymbol{\chi}) - \text{vectorize}(\boldsymbol{M})$, and $|\boldsymbol{\Omega}_i|$ denotes the determinant of $\boldsymbol{\Omega}_i$. To represent the distribution more simply, (3) gives an equivalent but a clearer formulation of (1):

$$\boldsymbol{\chi} \sim TN(\boldsymbol{M}, \boldsymbol{\Omega}_1, \boldsymbol{\Omega}_2, \ldots, \boldsymbol{\Omega}_k), \qquad (3)$$

2.3 Original Multi-Linear Relationship Network

Under the original MRN, there are m tasks in the task set $T = \{t_1, t_2, \ldots, t_m\}$, and for each $t_i \in T$, there are n_i exemplars. More specifically, each task t_i has sample pairs $(\mathbf{x}_j^{t_i}, \mathbf{y}_j^{t_i})$, where $j = 1,2,\ldots,n_i$. For task t_i, $\mathbf{x}_j^{t_i} \in \mathbb{R}^K$ is the input vector of length K, and $\mathbf{y}_j^{t_i} \in \{0,1,2,\ldots,C-1\}$ is the classification label (supervised learning) for $\mathbf{x}_j^{t_i}$. C represents the number of all distinct classes (labels) across all tasks.

Like most recent MTL models, there are shared feature layers and task-specific layers in MRN. For shared bottom layers (a pre-trained model), proposers of [32] do not place priors on any model parameter tensor. Instead, priors are used for task-specific (fully-connected) layers. When there is only one fully-connected layer, its parameter tensor is $\boldsymbol{W} = [\mathbf{w}_1; \mathbf{w}_2; \ldots; \mathbf{w}_m] \in \mathbb{R}^{d_1 \times d_2 \times m}$. $\mathbf{w}_i$ is the parameter matrix for the task t_i, and d_1 and d_2 denote the row and column numbers of $\mathbf{w}_i$. Namely, the shape of each $\mathbf{w}_i$ is the same.

In this project, since only a single task-specific layer is deployed under all circumstances, the maximum a posteriori (MAP) estimate (4) is slightly simpler than that in the original paper:

$$p(\boldsymbol{W}|\boldsymbol{\chi}, \boldsymbol{Y}) \propto p(\boldsymbol{W}) \prod_{i=1}^{m} \prod_{j=1}^{n_i} p\left(\mathbf{y}_j^{t_i} \middle| \mathbf{x}_j^{t_i}, \boldsymbol{W}\right), \qquad (4)$$

$\boldsymbol{\chi}$ and $\boldsymbol{\Upsilon}$ are the tensors containing all input samples and classification labels, respectively. As referred to in [32], $\prod_{i=1}^{m}\prod_{j=1}^{n_i} p\left(\mathbf{y}_j^{t_i}\middle|\mathbf{x}_j^{t_i}, \boldsymbol{W}\right)$ can be estimated with the empirical training loss (cross entropy loss) function. Accordingly, equation (3) can be used to describe $p(\boldsymbol{W})$:

$$p(\boldsymbol{W}) = TN_{d_1\times d_2\times m}(\boldsymbol{O}, \boldsymbol{\Omega}_1, \boldsymbol{\Omega}_2, \boldsymbol{\Omega}_3), \qquad (5)$$

As (5) denotes, the prior has the order of 3, and $\boldsymbol{O}$ is a zero tensor. $\boldsymbol{\Omega_1} \in \mathbb{R}^{d_1\times d_1}$, $\boldsymbol{\Omega_2} \in \mathbb{R}^{d_2\times d_2}$, and $\boldsymbol{\Omega}_3 \in \mathbb{R}^{m\times m}$ are the covariance matrices for modelling the relations of pairwise features, classes, and tasks, respectively.

Finally, the prior can be combined with the empirical training loss to form the objective function (6):

$$\min \sum_{i=1}^{m}\sum_{j=1}^{n_i} \mathrm{L}\left(F_i(\mathbf{x}_j^{t_i}), \mathbf{y}_j^{t_i}\right) + \frac{1}{2}(\text{vectorize}(\boldsymbol{W})^{\mathrm{T}}(\boldsymbol{\Omega}_1 \otimes \boldsymbol{\Omega}_2 \otimes \boldsymbol{\Omega}_3)^{-1}\text{vectorize}(\boldsymbol{W}) - \sum_{u=1}^{3}\frac{D_{total}}{D_u}\ln(|\boldsymbol{\Omega}_u|)), \qquad (6)$$

In (6), $\mathrm{L}(\cdot)$ means a cross entropy loss function with $F_i(\mathbf{x}_j^{t_i})$ representing the estimation of $\mathbf{y}_j^{t_i}$ produced by the composite model. The task-specific layer is fully-connected whose order of its parameter tensor $\boldsymbol{W}$ is only 3. Thus, $D_{total} = \prod_{v=1}^{3} D_v$, where $D_1 = d_1$, $D_2 = d_2$, and $D_3 = m$.

In the original paper, the loss function (6) is optimized by updating $\boldsymbol{W}$ when the covariance matrices are fixed and updating $\boldsymbol{\Omega}_1$, $\boldsymbol{\Omega}_2$, and $\boldsymbol{\Omega}_3$ when $\boldsymbol{W}$ is fixed. Accordingly, the term $\sum_{u=1}^{3}\frac{D_{total}}{D_u}\ln(|\boldsymbol{\Omega}_u|)$ is ignored during optimizing $\boldsymbol{W}$, and the updating process of covariance matrices is denoted in (7), (8), and (9):

$$\boldsymbol{\Omega}_1 = \frac{1}{D_2 D_3}\mathbf{W}_{(1)}(\boldsymbol{\Omega}_2 \otimes \boldsymbol{\Omega}_3)^{-1}\mathbf{W}_{(1)}^{\mathbf{T}} + \varepsilon\mathbf{I}_{D_1}, \qquad (7)$$
$$\boldsymbol{\Omega}_2 = \frac{1}{D_1 D_3}\mathbf{W}_{(2)}(\boldsymbol{\Omega}_1 \otimes \boldsymbol{\Omega}_3)^{-1}\mathbf{W}_{(2)}^{\mathbf{T}} + \varepsilon\mathbf{I}_{D_2}, \qquad (8)$$
$$\boldsymbol{\Omega}_3 = \frac{1}{D_1 D_2}\mathbf{W}_{(3)}(\boldsymbol{\Omega}_1 \otimes \boldsymbol{\Omega}_2)^{-1}\mathbf{W}_{(3)}^{\mathbf{T}} + \varepsilon\mathbf{I}_{D_3}, \qquad (9)$$

Where $\mathbf{W}_{(i)}$ means the matrix from applying the mode-i flattening to $\boldsymbol{W}$, and ε is a relatively small value for the sake of making the equations more stable. $\mathbf{I}_D$ is an identity matrix with width and height D. Next, by utilizing the properties of the matrix direct product, the equation can be further modified as the following equation:

$$(\boldsymbol{\Omega}_1)_{ij} = \frac{1}{D_2 D_3}\mathbf{W}_{(1)_i} \cdot \text{vectorize}(\boldsymbol{\Omega}_2{}^{-1}\boldsymbol{W}_{..j}\boldsymbol{\Omega}_3{}^{-1}) + \varepsilon\mathbf{I}_{D_1}, \qquad (10)$$

The equation (10) is proved to be equivalent to (7). $\mathbf{W}_{(1)_i}$ denotes the row with the index i from $\mathbf{W}_{(1)}$. $\boldsymbol{W}_{..j}$ means the $[:, :, j]$ slice gained from $\boldsymbol{W}$. To ensure that $\boldsymbol{\Omega_1}^{-1}$, $\boldsymbol{\Omega_2}^{-1}$, and $\boldsymbol{\Omega_3}^{-1}$ are existent, the researchers factorize each covariance matrix for keeping this constraint.

2.4 Bi-directional LSTM

The unit is composed of a forward LSTM [29] cell and a backward LSTM unit. How a single LSTM cell is formed is illustrated in Figure 2, and a cell has a forget gate, and gates for output and input, respectively. For updating an LSTM cell, the formulas listed below are applied:

$\mathbf{i_t} = \sigma(\mathbf{D_i x_t} + \mathbf{b_i} + \mathbf{W_i h_{t-1}})$, (11)
$\mathbf{f_t} = \sigma(\mathbf{D_f x_t} + \mathbf{b_f} + \mathbf{W_f h_{t-1}})$, (12)
$\mathbf{\hat{c}_t} = \tanh(\mathbf{D_c x_t} + \mathbf{b_c} + \mathbf{W_c h_{t-1}})$, (13)
$\mathbf{c_t} = \mathbf{i_t} \odot \mathbf{\hat{c}_t} + \mathbf{f_t} \odot \mathbf{c_{t-1}}$, (14)
$\mathbf{o_t} = \sigma(\mathbf{D_o x_t} + \mathbf{b_o} + \mathbf{W_o h_{t-1}})$, (15)
$\mathbf{h_t} = \mathbf{o_t} \tanh(\mathbf{c_t})$, (16)

Where $\mathbf{i_t}$ is the information for remembering, $\mathbf{x_t}$ indicates the current input token representation, $\mathbf{f_t}$ is the result from the forget gate, $\mathbf{\hat{c}_t}$ and $\mathbf{c_t}$ mean the temporary information of state and the updated state information, respectively. $\mathbf{o_t}$ indicates the output gate result, and $\mathbf{h_t}$ denotes the output of the whole cell. $\odot$ and $\sigma(\cdot)$ are product operators and sigmoid functions calculating element-wisely. $\tanh(\cdot)$ is a hyperbolic tangent activation function. $\mathbf{W_i}$, $\mathbf{W_f}$, $\mathbf{W_c}$, $\mathbf{W_o}$ are the parameters for the hidden layer, and $\mathbf{D_i}$, $\mathbf{D_f}$, $\mathbf{D_c}$, $\mathbf{D_o}$ are the parameters for the input vector, and $\mathbf{b_i}$, $\mathbf{b_f}$, $\mathbf{b_c}$, $\mathbf{b_o}$ are biases.

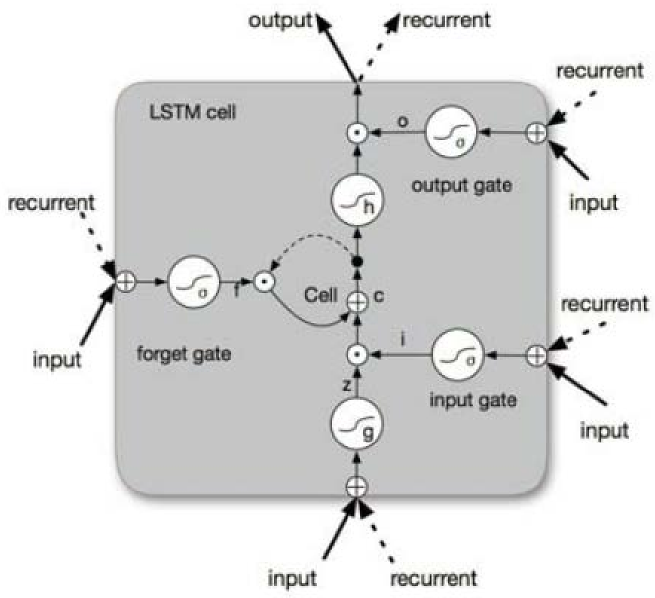


Figure 2: How an LSTM cell is formed [10].

The structure of a bi-directional LSTM (BiLSTM) unit is shown in Figure 3, and how two LSTMs are combined is given below:

$\mathbf{h_t^b} = \mathbf{LSTM^b}(\mathbf{x_t}, \mathbf{h_{t+1}^b})$, (17)
$\mathbf{h_t^f} = \mathbf{LSTM^f}(\mathbf{x_t}, \mathbf{h_{t-1}^f})$, (18)
$\mathbf{u_t} = \tanh(\mathbf{W_b h_t^b} + \mathbf{W_f h_t^f})$, (19)
$\mathbf{y_t} = \mathrm{softmax}(\mathbf{W_y u_t})$, (20)

Where $\mathbf{h_t^b}$ and $\mathbf{h_t^f}$ are hidden states of the two LSTMs: $\mathbf{LSTM^b}$ (backward) and $\mathbf{LSTM^f}$ (forward). $\mathbf{W_b}$, $\mathbf{W_f}$ and $\mathbf{W_y}$ are their weight matrices. $\mathbf{u_t}$ is the combined information of the LSTMs, and $\mathbf{y_t}$ is the final output label distribution of the model. softmax$(\cdot)$ is a softmax transfer function.

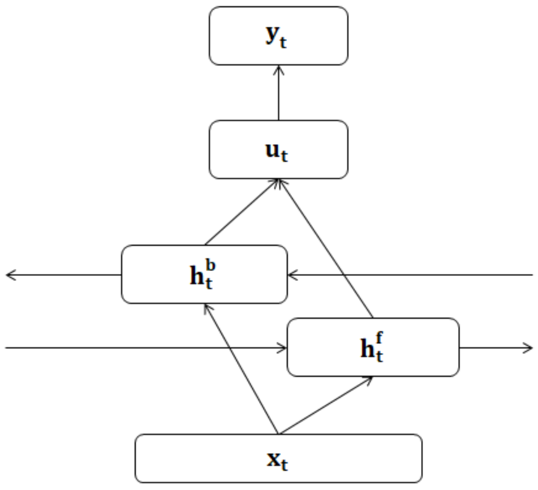


Figure 3: The Structure of a BiLSTM unit.

2.5 BERT

BERT is an abbreviation of “Bidirectional Encoder Representations from Transformers”, and a BERT encoder has multiple layers having the multi-head-attention [13] mechanism, which alleviates the long-term dependence problem that typical Recurrent Neural Networks (RNNs) have.

As [7] mentioned, BERT needs a pre-training process which is time-consuming, so in the project, a pre-trained BERT$_{base}$ model is loaded as a word embedding encoder. Besides, only a brief workflow of how BERT copes with sequence labelling tasks is presented. Namely, the detail of the mechanism in a BERT model is not discussed. The simple workflow is shown in Figure 4, where input sentences must be converted by a pre-trained tokenizer to form a list or array of tokens. Meanwhile, the tokenizer generates token embeddings, and after this, the embedding vectors are passed through the model to obtain the hidden output. Note that an additional layer has to be added for a token-level task like sequence labelling to acquire the final result.

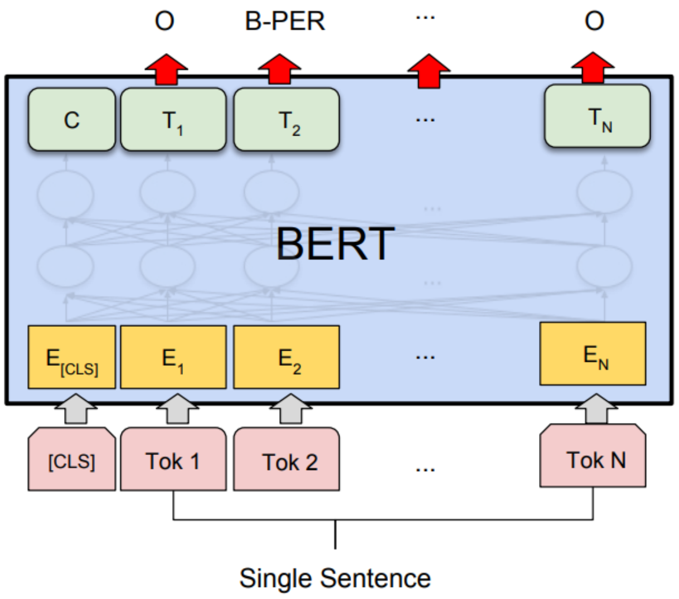


Figure 4: A simple workflow of BERT dealing with a single sequence labelling task from [7]. [CLS] is the token for the start of the sequence. $Tok\ i$ means the original i-th input token. E_i indicates the sum of

(segmentation, token, and position) embeddings generated by a tokenizer in advance for the corresponding token, and $\mathbf{T}_i$ is the hidden output of BERT for $Tok\ i$. The labels on the top of the figure are the final output labels for the task, and they are often produced from a single linear layer.

A pre-trained BERT$_{\text{base}}$ model has 110 million model parameters, a hidden size of 768, 12 Transformer blocks and 12 attention heads. Moreover, there are two pre-training steps of BERT using BookCorpus [40] and the English version of Wikipedia:

1. Mask LM:

   "LM" here refers to "Language Modelling". In this step, before the tokens are sent to BERT, some tokens are masked randomly in order to make BERT guess what the masked tokens are. Typically, a mask token "[MASK]" is used to pre-train the model.

2. Next Sentence Prediction:

   In this task (NSP), BERT is pre-trained to learn the relations of two sentences. In other words, the model can learn which sentence is more probable to follow the target sentence. Normally, a [CLS] token is placed at the start of the input sentences. A [SEP] token is then placed to separate two sentences.

After BERT is pre-trained, it can then be fine-tuned according to the strategies which will be discussed in Section 2.7.

2.6 RoBERTa

In the paper [18], a BERT model is "robustly optimized" with a variety of strategies in the pre-training steps to become RoBERTa. A pre-trained RoBERTa$_{\text{base}}$ model has 125 million trainable model parameters, a hidden size of 768, 12 Transformer blocks, and 12 attention heads. Like in Section 2.5, only a pre-trained RoBERTa$_{\text{base}}$ model is directly downloaded to be a word embedding encoder in this project. RoBERTa$_{\text{base}}$ is not pre-trained from scratch, and the detail of the mechanism in a RoBERTa model is not mentioned. Except for the pre-training datasets used for BERT, there are more pre-training datasets for RoBERTa, which are Stories [42], OpenWebText [41] and CC-News [43]. Besides, the pre-training approaches different from the original BERT paper [7] are listed as the following:

1. Dynamic Masking:

   The original BERT model is pre-trained with static (random) masking. By contrast, the masking step for RoBERTa is dynamic, which means a distinct pattern of masking is created when new tokens are fed.

2. No NSP loss:

   The paper claims that NSP does not enhance the abilities of the models for target fine-tuning tasks. In other words, NSP may be too easy for target tasks coming later in the fine-tuning steps resulting in a drop in performance. Accordingly, RoBERTa is not pre-trained using NSP.

3. Larger Mini-batch Size:

   RoBERTa is trained with an increased size of mini-batch of 8K, while BERT is trained using smaller mini-batches of size 256. [18] shows that this leads to the improvement of the model performance.

4. Greater Byte-level Byte-Pair Encoding:

   Unlike the character-level BPE used for the original BERT model, RoBERTa is trained using BPE that is byte-level [30]. The sizes of vocabulary for pre-training steps of BERT and RoBERTa are 30K and 50K, respectively. As a result, a RoBERTa$_{\text{base}}$ model has 15 million trainable parameters more than those of BERT$_{\text{base}}$ due to an increase in vocabulary size.

2.7 Fine-tuning Strategies

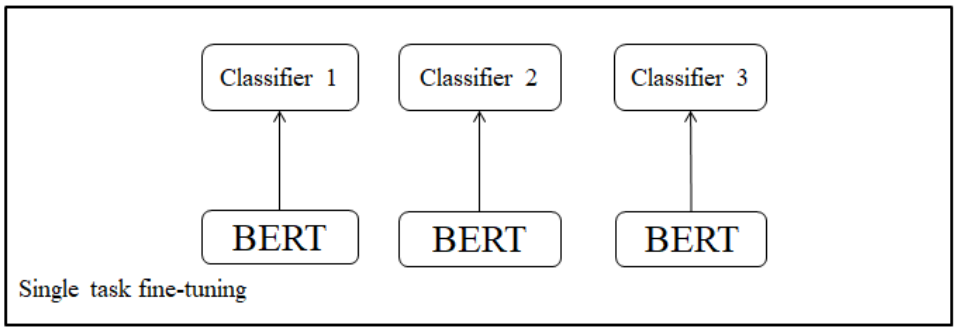
The fine-tuning step for BERT-like models is a part of transfer learning [31]. After a model is pre-trained, the weights and biases of the model are inherited for some learning new target tasks that are normally related to the former ones. In the fine-tuning step, a BERT-like model is usually concatenated by one or more additional layers such as linear layers, and then the parameters of the whole model are tuned using target datasets using some gradient descent methods. Furthermore, researchers in [20] explore several important strategies to fine-tune a Transformer encoder for multiple tasks, and Figure 5 gives a brief overall picture of them:

1. Single Task Fine-tuning:

This strategy is identical to STL. When there are several tasks, individual models are used for each task. A benefit of the approach is that all models are trained separately, so tasks are immune to one another. Namely, if a task is harmful to others, it cannot influence the performance of others. However, this method does not support any parameter-sharing approach. The total number of parameters of the models increases dramatically if there are multiple tasks due to the large size of a BERT-like model.

2. Multiple Tasks Fine-tuning:

This fine-tuning method (called “Multi-Task Fine-tuning” in [20]) aims to share bottom layers and train the whole model with a specified loss function like the sum of cross entropy loss functions for Multi-Task sequence labelling. This strategy is the same as training a composite model by sharing the bottom layers in MTL. Accordingly, the model is composed of one or more shared layers at the bottom and a task-specific layer at the top (hard parameter sharing). The proportions of loss functions for various tasks are normally the same.



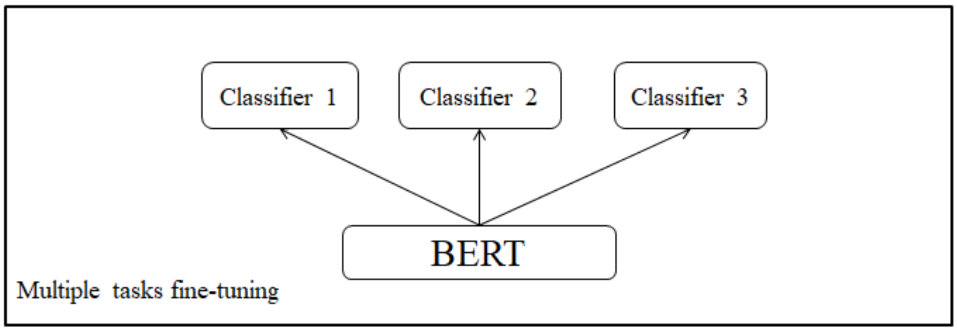


Figure 5: An overall picture of the two main fine-tuning approaches, taking BERT as an example. Note that each classifier is for a target task.

3 Objectives, Specifications and Design

This section will provide the recall of objectives together with project requirements and specifications. The detailed design of the workflow for achieving the goals of the project will also be shown.

3.1 Objectives Review

The project aims to introduce MRN to sequence labelling. In the first chapter, the objectives of the project were briefly addressed, and the detailed definition of a Multi-Task sequence labelling problem was described next. Accordingly, in this section, the objectives will be reviewed and extended.

First of all, the original MRN mentioned earlier is still required to be modified for a Multi-Task sequence labelling problem even if MRN can cope with multiple multi-class classification tasks which share the same classes. When some classes are different between tasks, the original MRN cannot place a prior on the parameter tensor of the classifier layer. Except for this, when all tasks share the same classes in the original MRN, the prior can be put on the model parameter tensor of a single task-specific (classifier) layer (composed of several linear layers), and this is also applicable to a Transformer-based model where a task-specific layer is deployed. Accordingly, the only challenge left is how to arrange the weight tensors for different or the same types of labels across all sequence labelling tasks. In this project, the proposed way to achieve this is simply aligning the weight tensors used for the same or different labels across all tasks. For example, it is assumed that there are 2 different POS tagging tasks l_1 and l_2 with 3 kinds and 2 kinds of labels, respectively, and there are 2 kinds of labels exactly the same (all kinds of labels in the second task are included in the first task). The shapes of the parameter tensors are [1, 3, d_1] for l_1 and [1, 2, d_1] for l_2, where d_1 is the number of input features from the shared layers. Then, for aligning the weight tensors, the [:, 0, :] tensor slice in the weight tensor of l_1 is for the first common label, and the [:, 0, :] tensor slice in the weight tensor of l_2 is also for the first common label. Likewise, the [:, 1, :] tensor slices in the two parameter tensors are aligned for the second common label. Finally, the weight tensor for l_2 is appended zeros to ensure that both of the two tensors have the shape [1, 3, d_1] so that the final combined parameter tensor for the task-specific layer has the shape [2, 3, d_1]. Namely, the final shape must be [d_3, d_2, d_1]. d_2 denotes the number of distinct types of labels across all sequence labelling tasks, and d_3 indicates the number of sequence labelling tasks. In short, the tensor alignment approach can ensure each weight tensor for the corresponding sequence labelling task has the shape [1, d_2,

d_1] so that the tensors can be combined for calculating the regularization loss and updating the covariance matrices.

Before applying the modified MRN to Multi-Task sequence labelling, each sequence labelling dataset should be divided into at least 2 splits for training and testing. More specifically, there are m tasks in the sequence labelling task set $L = \{l_1, l_2, \ldots, l_m\}$, and there are n_i sequences for each $l_i \in L$. Then, the training set for l_i can be $\{\boldsymbol{\chi}_{train}^{l_i}, \boldsymbol{Y}_{train}^{l_i}\}$. $\boldsymbol{\chi}_{train}^{l_i}$ contains sample sentences $\mathbf{x}_j^{l_i}$, and $\boldsymbol{Y}_{train}^{l_i}$ has the corresponding label sequences $\mathbf{y}_j^{l_i}$, where $j = 1,2,\ldots,n_i$. $\mathbf{x}_j^{l_i}$ can also be seen as a list of tokens with the maximal length K, and each element in $\mathbf{y}_j^{l_i}$ is an integer from $\{0,1,2,\ldots,C-1\}$ representing a label. C is the total number of types of distinct labels in all tasks. Likewise, the testing set for l_i can be $\{\boldsymbol{\chi}_{test}^{l_i}, \boldsymbol{Y}_{test}^{l_i}\}$, and the rest of the expressions are similar to the training split.

When the datasets are prepared, the pre-trained Transformer encoder models with their tokenizers should be loaded for building the composite models. In this project, the shared layers at the bottom of the composite models are BERTbase, RoBERTabase, and BERTbase-BiLSTM, and the structure of the composite models will be shown in Chapter 5. Next, the composite model can be fine-tuned (trained) for several epochs with a specified loss function under STL, MTL or MRN. For MRN, an extra regularization loss is added to the loss function, and it can be calculated with the combined parameter tensor referred to earlier. Besides, the AdamW [33] optimizer is chosen for fine-tuning the composite models.

The fine-tuned composite model should be tested with the testing split after it is trained, and the project then seeks to compare the performance of various composite models under different learning methods. Because of this, how the models are evaluated is given here:

1. Compare different Transformer-based models under an identical learning paradigm:
   In this project, when different Transformer-based models are trained under an identical learning method (STL, MTL or MRN), their F1-scores on the same dataset are compared to see which model outperforms the others.

2. Compare different learning paradigms by the performance of the same Transformer-based model:
   The same Transformer-based model is trained under STL, MTL or MRN to see under which learning method the model performs better using F1-scores on the same dataset.

In addition, to see if MRN can force the models to learn the task covariance, the covariance matrix can be visualized with a map. The colours in the map should specify the positive, zero, and negative relations. With the review and extensions of the objectives, the requirements together with several technical specifications will be provided in the next subsection.

3.2 Requirements and Specifications

To execute the objectives, some requirements are necessary, and specifications have to be elaborated for implementation. Since the Transformer-based models must be fine-tuned with faster GPUs, Google Colaboratory [34] is chosen as the platform for implementation.

3.2.1 Requirements

- Implementing the relevant tensor alignment functions to modify the original MRN;
- Loading and pre-processing the benchmark datasets;

- Dividing the pre-processed dataset into separate sets with specified batch sizes for training and testing (or validation when needed);
- Loading pre-trained Transformer encoders together with their tokenizers and building the composite models;
- Implementing the loss functions in the training step and adding the regularization term if models are trained under MRN for sequence labelling;
- Saving the models during training;
- Tuning the (hyper) parameters when needed;
- Successfully making predictions for each sequence labelling task under STL, MTL or MRN;
- Evaluating the predictions from the models with the approaches mentioned in Section 3.1;
- Visualizing the relations that MRN models learn.

3.2.2 Specifications

- Platform: Google Colaboratory (Pro version)
- Language: Python 3.7.13
- Core Imported Modules: Pytorch [35], Seqeval, Truecase, Transformers, Datasets, etc.
- GPU: Tesla P100-PCIE-16GB

3.3 Design

In this section, the design related to the overall workflow of the project will be described, and the ways to optimize the model under different learning paradigms will be elaborated.

3.3.1 Overview

The overall workflow can be divided into 2 parts, including the training stage and the evaluation stage. As Figure 6 illustrates, before a batch of input sequences are sent into the sequence labelling model, they are first pre-processed by a pre-trained tokenizer to produce token, position and segment representations. Then, the embeddings are loaded by a composite Transformer-based model, and the output tensors are compared with true labels of the corresponding sequence labelling tasks to calculate task-specific losses. How the losses are combined will be shown in Section 3.3.2, and the formulation of the total loss varies because of different learning methods. For instance, when MRN is not applied, the second term (the regularization term in (6) (including $L_{reg} = (\text{vectorize}(\boldsymbol{W})^{\mathrm{T}}(\boldsymbol{\Omega}_1 \otimes \boldsymbol{\Omega}_2 \otimes \boldsymbol{\Omega}_3)^{-1}\text{vectorize}(\boldsymbol{W}))$) is ignored. When the total loss is calculated, the AdamW optimizer can tune all the trainable parameters. For MRN, the procedure of updating the covariance matrices follows the original methods in Section 2.3. After the optimizer fine-tunes the model for several iterations, the covariance matrices are updated. Namely, the task, class, and feature covariance matrices are updated with a specified frequency.

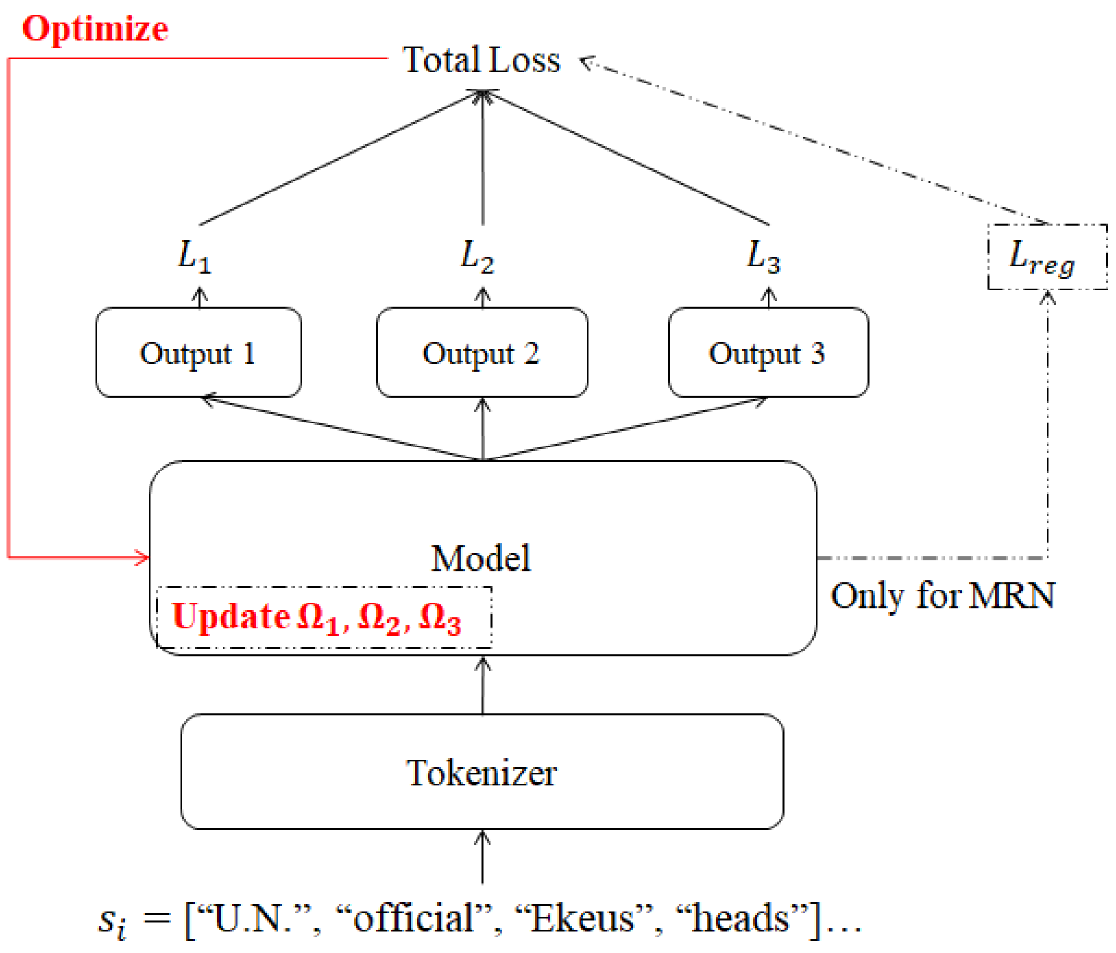


Figure 6: The overall view of the training stage for 3 sequence labelling tasks. s_i is an input sequence from a single batch of the training dataset. L_1, L_2, and L_3 are the cross entropy losses of the corresponding sequence labelling tasks, and L_{reg} is the regularization term only for MRN. Besides, for MRN, the covariance matrices are updated with a specified frequency during fine-tuning.

After the model is fine-tuned, its performance should be evaluated with the testing dataset. There are ground-truth labels for each sequence labelling task in the dataset for testing. Therefore, the F1-score on each task can be calculated for evaluation. The simplified concepts of how to evaluate the models under different learning methods were listed in Section 3.1.

3.3.2 Learning Paradigms

The basic strategies for fine-tuning were referred to in Section 2.7, and there are 3 ways for optimizing the composite model under different learning paradigms in the project. In a word, the equations below are designed based on STL, MTL or MRN. L_t is a cross entropy loss function for the t-th sequence labelling task. $\boldsymbol{W}$ denotes the parameter tensor for the task-specific layer in the model. The "task-specific layer" here also means the classifier layer which is composed of all linear layers used for corresponding sequence labelling tasks.

- STL:

Equivalent to single task fine-tuning, since there is only one target sequence labelling task in this scenario, the model is trained based on a single loss. A model is trained for a sequence labelling task, which means multiple models are trained for multiple tasks under STL.

$$\min(L_1), \tag{21}$$

- MTL:

The same as a basic multiple tasks fine-tuning approach, the function (22) means the summation of the m losses for m sequence labelling tasks. The whole model is fine-tuned under this criterion, and tasks are treated equally. In this case, a model is trained for multiple tasks.

$$\min(\textstyle\sum_{t=1}^{m} L_t), \tag{22}$$

- MRN:

As mentioned in Section 2.3, $\sum_{u=1}^{3} \frac{D_{total}}{D_u} \ln(|\mathbf{\Omega}_u|)$ is ignored during optimizing $\boldsymbol{W}$, and the rules for optimizing covariance matrices are also given in the section. Therefore, the objective function (22) can be extended and integrated as the optimization procedure below (each $\mathbf{w}_i$ in $\boldsymbol{W}$ needs to be aligned in advance):

$$L_{total} = \textstyle\sum_{t=1}^{m} L_t + \lambda \cdot L_{reg}, \tag{23}$$
$$(\mathbf{\Omega}_1)_{ij} = \frac{1}{D_2 D_3} \mathbf{W}_{(1)_i} \cdot \mathrm{vectorize}(\mathbf{\Omega}_2{}^{-\mathbf{1}} \boldsymbol{W}_{..j} \mathbf{\Omega}_3{}^{-\mathbf{1}}) + \varepsilon \mathbf{I}_{D_1}, \tag{24}$$
$$(\mathbf{\Omega}_2)_{ij} = \frac{1}{D_1 D_3} \mathbf{W}_{(2)_i} \cdot \mathrm{vectorize}(\mathbf{\Omega}_1{}^{-\mathbf{1}} \boldsymbol{W}_{..j} \mathbf{\Omega}_3{}^{-\mathbf{1}}) + \varepsilon \mathbf{I}_{D_2}, \tag{25}$$
$$(\mathbf{\Omega}_3)_{ij} = \frac{1}{D_1 D_2} \mathbf{W}_{(3)_i} \cdot \mathrm{vectorize}(\mathbf{\Omega}_1{}^{-\mathbf{1}} \boldsymbol{W}_{..j} \mathbf{\Omega}_2{}^{-\mathbf{1}}) + \varepsilon \mathbf{I}_{D_3}, \tag{26}$$

Where $L_{reg} = (\mathrm{vectorize}(\boldsymbol{W})^{\mathrm{T}} (\mathbf{\Omega}_1 \otimes \mathbf{\Omega}_2 \otimes \mathbf{\Omega}_3)^{-1} \mathrm{vectorize}(\boldsymbol{W}))$ is the regularization loss with the hyper-parameter λ for deciding the proportion of the regularizer. $\mathbf{\Omega}_1$, $\mathbf{\Omega}_2$, and $\mathbf{\Omega}_3$, are covariance matrices for modelling feature, class (label), and task covariance, respectively. Overall, MRN seeks to minimize L_{total} indicating the total loss, and then it updates the covariance matrices with a certain frequency in the fine-tuning stage.

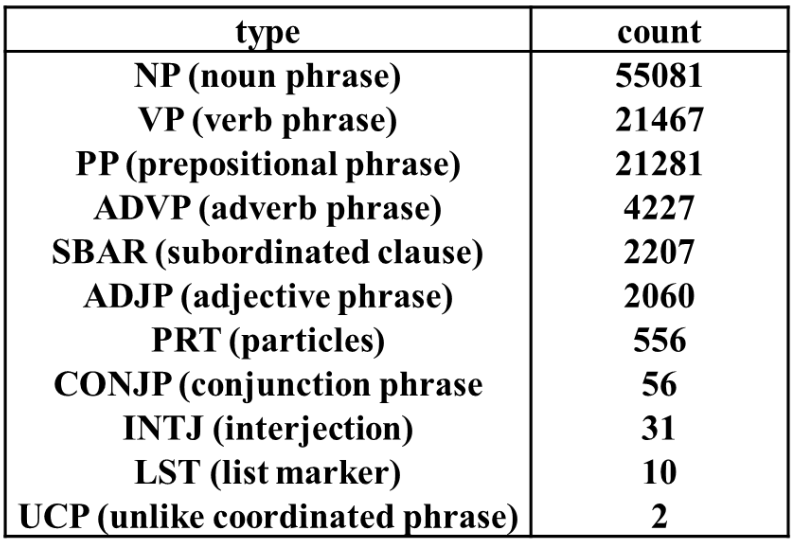
4 Datasets

The benchmark sequence labelling datasets used in this project are the English versions of CoNLL-2000 and CoNLL-2003. These datasets contain tokens from articles together with their classification labels of the corresponding sequence labelling tasks.

4.1 CoNLL-2000

CoNLL-2000 was produced from WSJ [36], and there are 2 sequence labelling tasks in the dataset, which are chunking and POS tagging. The main task is text chunking where 22 labels are included in total; on the contrary, there are 44 labels for POS tagging. Besides, there is no overlapping label across all tasks, which means all labels are different from one another. CoNLL-2000 contains 211727 tokens in the training split and 47377 tokens in the testing set. The chunking labels are given in Figure 7. Note that each type of label in the figure has 2 variants (“I-“ and “B-“). For example, “I-NP” and “B-NP” belong to the type “NP”.

In the project, all the training samples are used in the training stage, and all the samples in the testing split are used for evaluation.



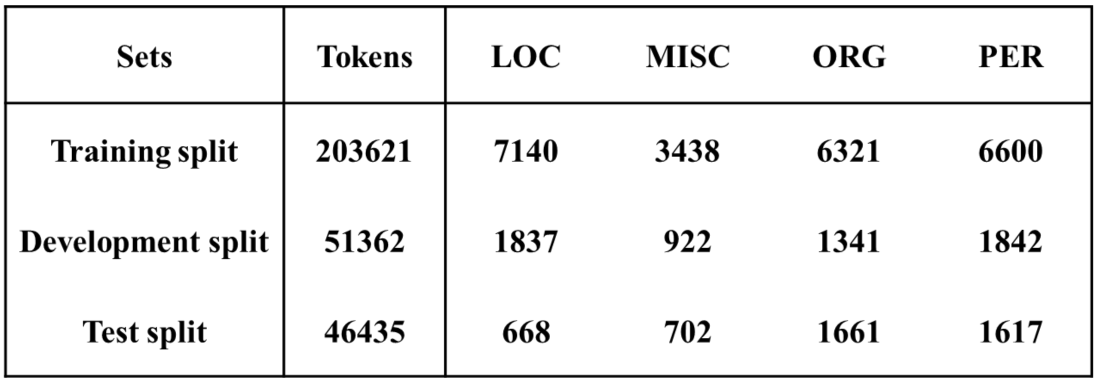
type	count
NP (noun phrase)	55081
VP (verb phrase)	21467
PP (prepositional phrase)	21281
ADVP (adverb phrase)	4227
SBAR (subordinated clause)	2207
ADJP (adjective phrase)	2060
PRT (particles)	556
CONJP (conjunction phrase	56
INTJ (interjection)	31
LST (list marker)	10
UCP (unlike coordinated phrase)	2

Figure 7: The labels for text chunking together with their counts in the training split of CoNLL-2000.

4.2 CoNLL-2003

CoNLL-2003 was produced from Reuters Corpus [25]. NER, text chunking and POS tagging labels are contained in the dataset. The main task is NER, which includes 8 labels in total. POS tagging and chunking have 47 and 22 labels, respectively. In addition, all labels in all tasks are distinct from one another, and Figure 8 shows that CoNLL-2003 contains 203621 tokens in the training set, 46435 tokens in the testing split and 51362 tokens for validation. Like CoNLL-2000, each type of label in the figure has 2 variants (“I-” and “B-”). For instance, “I-LOC” and “B-LOC” belong to the type “LOC”.

In the project, all of the training samples are used in the training stage, but the development samples are not directly used for training. The development split is only utilized for deciding whether to stop training the models to prevent over-fitting. The testing set is used for evaluation.



Sets	Tokens	LOC	MISC	ORG	PER
Training split	203621	7140	3438	6321	6600
Development split	51362	1837	922	1341	1842
Test split	46435	668	702	1661	1617

Figure 8: The labels for NER in CoNLL-2003. There are 4 types of named entities in total. (“LOC”: location, “MISC”: miscellaneous, “ORG”: organization, “PER”: person)

5 Methodology and Implementation

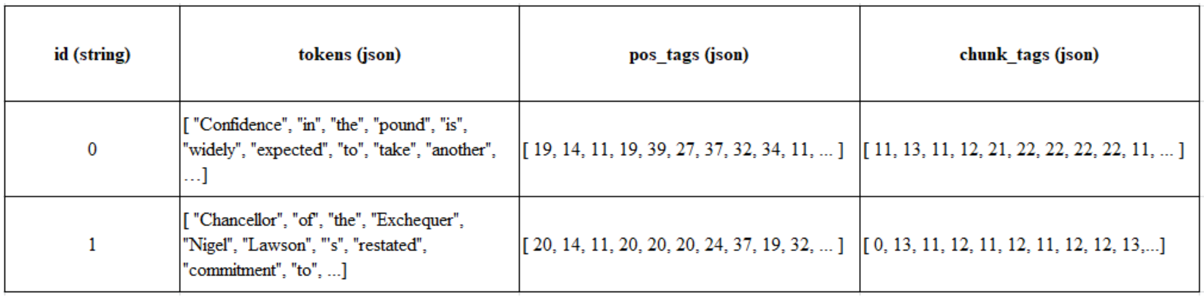
The fifth chapter will be focused on the methodology and its implementation for achieving the objectives. The following sections will present the explicit procedures for the experiment and the implementation. The specifications for implementation were listed in Section 3.2.2. The detailed implementation code will be given in the tenth chapter.

5.1 Datasets Loading and Processing

Before the datasets are sent into a tokenizer, they are required to be processed first. In this project, since the pre-trained BERT model is case-sensitive, all the uppercase sentences in the datasets are transformed by truecasing [38]. Following this, the datasets are sent to the tokenizer.

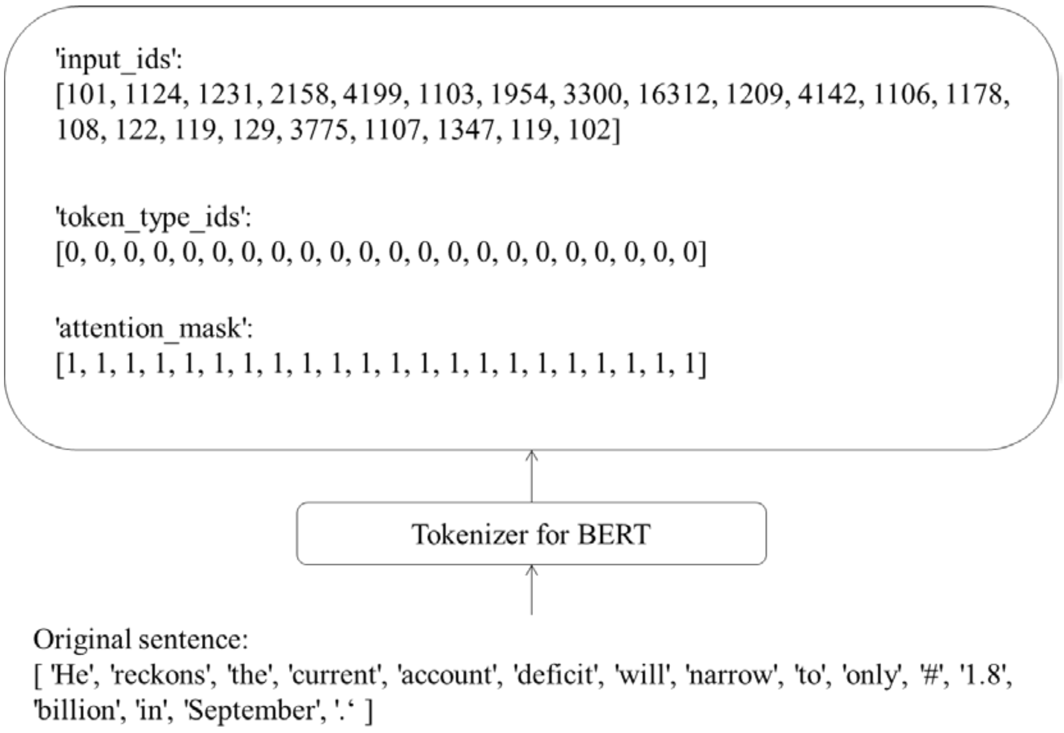
After the original sentences are truecased and tokenized, the true labels also need to be aligned with the output representations from the tokenizer. In other words, some tokens in the original sentence are further tokenized into several tokens, so extra labels are required to represent the original tokens. For instance, if the token “reckons” is divided into “re”, “##ck”, and “##ons”, the original true chunking label “21” for “reckons” is not sufficient to represent the whole word. Therefore, extra labels “-100” and “-100” are added for the “##ck” and “##ons” tokens, respectively. Consequently, the tokens “re”, “##ck”, and “##ons” correspond to the true labels “21”, “-100”, and “-100”. Besides, in the case of BERT, the [CLS] tokens and [SEP] tokens are also labelled “-100” for corresponding ground-truth labels. Note that the label “-100” denotes that the corresponding token is not interesting for sequence labelling. Finally, the processed datasets are all well-prepared and ready to be sent into the composite model.

The CoNLL-2000 and CoNLL-2003 datasets are downloaded from Huggingface’s datasets [37], and the formats of the two datasets are almost the same except that CoNLL-2003 has extra NER labels (ner_tags). An example view of the raw CoNLL-2000 training dataset is shown in Figure 9. The tokenizers are directly downloaded from Huggingface's transformers [39], and they are responsible for tokenizing the sentences and producing vectors. As Figure 10 illustrates, the tokenizer for BERT can produce “input_ids”, “token_type_ids” and “attention_mask”, indicating the basic representations of tokens (token embeddings which are not ground-truth labels), specifying the split of the sentence, and whether masks are applied on the tokens. All the implementation code for processing and loading datasets will be shown in A.4 (like “truecase_tokens”, “tokenization_with_alignment”, “CoNLL_00”, and “CoNLL_03”).



id (string)	tokens (json)	pos_tags (json)	chunk_tags (json)
0	["Confidence", "in", "the", "pound", "is", "widely", "expected", "to", "take", "another", …]	[19, 14, 11, 19, 39, 27, 37, 32, 34, 11, ...]	[11, 13, 11, 12, 21, 22, 22, 22, 22, 11, ...]
1	["Chancellor", "of", "the", "Exchequer", "Nigel", "Lawson", "'s", "restated", "commitment", "to", ...]	[20, 14, 11, 20, 20, 20, 24, 37, 19, 32, ...]	[0, 13, 11, 12, 11, 12, 11, 12, 12, 13,...]

Figure 9: An example view of the raw CoNLL-2000 training set. The labels are displayed with the corresponding integers.



```
'input_ids':
[101, 1124, 1231, 2158, 4199, 1103, 1954, 3300, 16312, 1209, 4142, 1106, 1178,
108, 122, 119, 129, 3775, 1107, 1347, 119, 102]

'token_type_ids':
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

'attention_mask':
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Original sentence:
['He', 'reckons', 'the', 'current', 'account', 'deficit', 'will', 'narrow', 'to', 'only', '#', '1.8', 'billion', 'in', 'September', '.']

Figure 10: The output of a tokenizer, taking BERT as an example. In “input_ids”, the numbers “101” and “102” represent the [CLS] and the [SEP] tokens.

5.2 Models

Regarding the structure of the models, there are 3 types of composite sequence labelling models in the project. All of these models contain shared layers and a task-specific layer, which are BERT$_{base}$-Linear, RoBERTa$_{base}$-Linear, and BERT$_{base}$-BiLSTM-Linear. Note that the Transformer encoders with their tokenizers were all pre-trained in advance, so they are all downloaded, used directly and not pre-trained from scratch in the project. The structure of the models is shown in Figure 11. “-Linear” denotes that the shared Transformer-based encoder is concatenated by linear (fully-connected) layers. Linear layers are deployed as a task-specific layer for corresponding sequence labelling tasks. Accordingly, the number of sequence labelling tasks equals the number of linear layers.

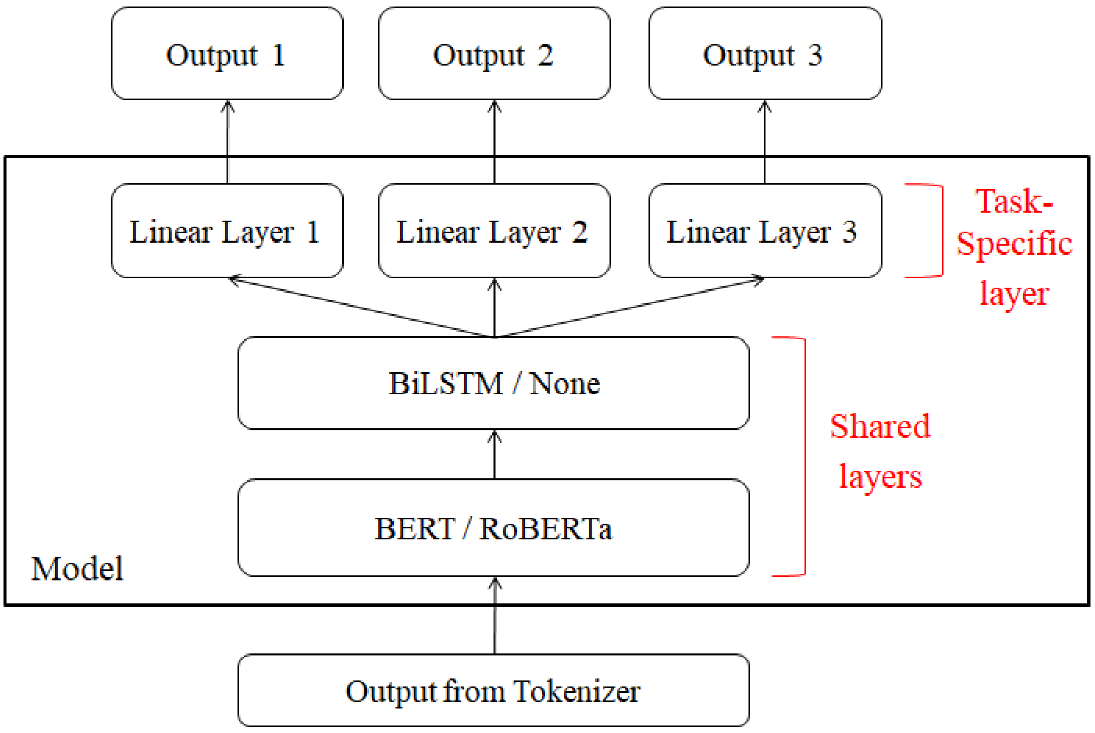


Figure 11: The structure of the models in the project under MTL or MRN. The shared layers can be BERTbase, RoBERTabase, or BERTbase-BiLSTM, and the task-specific layer is formed by all linear layers customized for each task. Consequently, there are 3 types of composite models in total. In addition, when a model is fine-tuned under STL, only one linear layer is deployed.

After a composite model is fine-tuned for a specific number of epochs, it can be evaluated in terms of the quality of its predictions. Hence, the model must output sequences of labels for corresponding sequence labelling tasks. Because of this, the output from the model for a token has to be transformed into an integer (index) representing a label. The methodology for label predictions is illustrated in Figure 12. Like in the training step, the composite model is fed with the embeddings produced from the tokenizer. However, in the testing step, a softmax function is used for normalization so that the predictions in each sequence labelling task sum up to 1. Following this, an argmax function outputs the labels in each task.

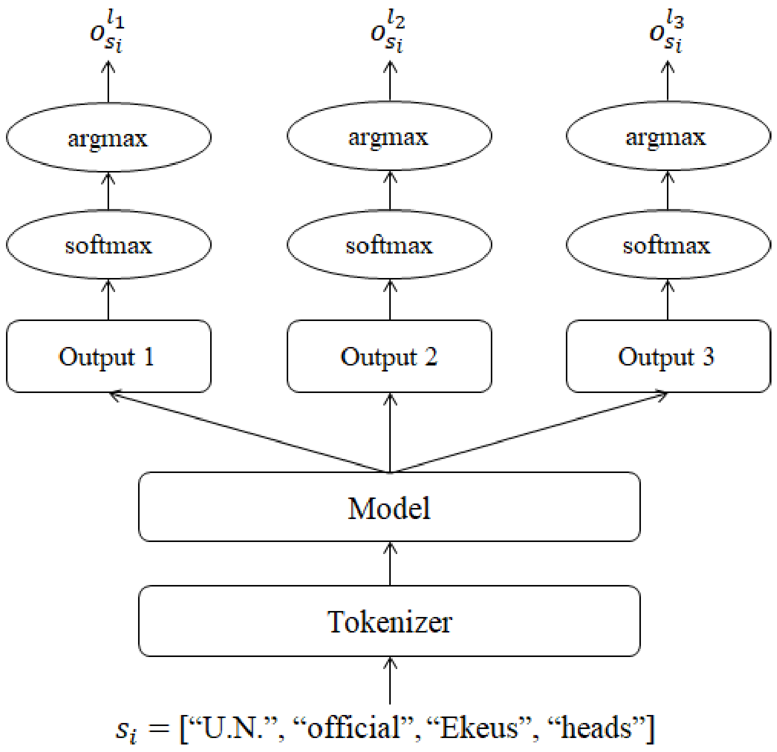


Figure 12: The methodology for label predictions in the testing stage. Assume a sequence set $S = \{s_1, s_2, \dots, s_n\}$ is a batch from the testing dataset. n means the number of sequences in the batch. $s_i \in S$ is a sequence of input tokens from the batch. l_1, l_2 , $l_3 \in L$ are sequence labelling tasks. $o_{s_i}^{l_1}$, $o_{s_i}^{l_2}$ and $o_{s_i}^{l_3}$ are the output sequences of labels for s_i in each task.

With the acquired labels, the F1-scores on each sequence labelling task can be calculated. Since the ground-truth labels of all the tasks are given in the dataset, the predictions can be compared with true labels directly for model evaluation.

As for the implementation of the composite models, pre-trained BERT$_{\text{base}}$ and RoBERTa$_{\text{base}}$ Pytorch models are downloaded from Huggingface's transformers, and BERT$_{\text{base}}$ is the "cased" version. The composite models can be built with "torch.nn.Module" as the bases. Besides, the structure of the composite models is implemented in A.3 (like the class "ROBERTA_MRN_conll2003"). The code for training and evaluating the models with certain datasets under different learning paradigms is implemented in A.5 (such as the function "train_conll2003_MRN") and A.6 (like the function "test_conll2003_MRN_or_MTL").

5.3 Weight Tensor Alignment for MRN

As mentioned in Section 3.1, it is necessary to align the weight tensors for each linear layer. This proposed approach is required only if the model is trained under MRN, and it is used for calculating L_{reg} and updating the covariance matrices. In this project, with the symmetric model structure, the property that the input dimensions (number of input features) of all linear layers are the same is utilized, which makes the alignment process easier. The only key point is to align the rows of weight tensors of all linear layers. If tasks share some labels, the weight tensors for the corresponding task are required to be aligned with one another. In this case, Figure 13 illustrates how the tensors are aligned. Each $\mathbf{w}_i$ is the weight tensor of the linear layer for the corresponding sequence labelling task, and in the example, each row in $\mathbf{w}_i$ represents the weight tensor for a corresponding label (the columns in $\mathbf{w}_i$ are for features). The concatenation of the weight tensors of all linear layers is equivalent to the task-specific model parameter tensor $\boldsymbol{W}$. Namely, $\boldsymbol{W}$ is the concatenation of all $\mathbf{w}_i$.

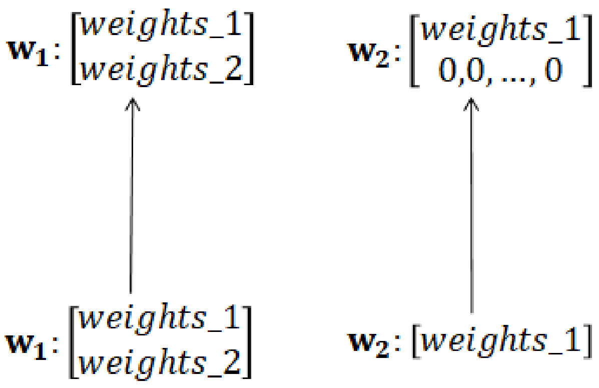

$\mathbf{w_1}: \begin{bmatrix} weights\_1 \\ weights\_2 \end{bmatrix}$
$\mathbf{w_2}: \begin{bmatrix} weights\_1 \\ 0,0,\ldots,0 \end{bmatrix}$
$\mathbf{w_1}: \begin{bmatrix} weights\_1 \\ weights\_2 \end{bmatrix}$
$\mathbf{w_2}: [weights\_1]$

Figure 13: The transformation of $\mathbf{w}_i$ in $\boldsymbol{W}$ when some labels are the same in the tasks. In the example, there are 2 sequence labelling tasks, and it is assumed that the second task has exactly one label the same as the first task. There are 2 distinct labels across all tasks, and $\mathbf{w}_2$ is appended 0 to ensure that both of the tensors have the same dimension.

By contrast, when the output spaces of sequence labelling tasks are all distinct, the output spaces are combined into a general output space in this project. Namely, as Figure 14 illustrates, to ensure each $\mathbf{w}_i$ in the task-specific parameter tensor $\boldsymbol{W}$ has the same dimensions, $\mathbf{w}_i$ is appended 0 if needed.

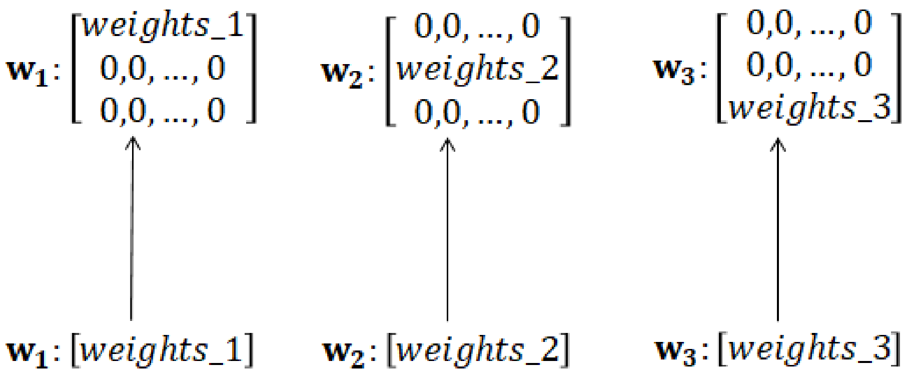

$\mathbf{w_1}: \begin{bmatrix} weights\_1 \\ 0,0,\ldots,0 \\ 0,0,\ldots,0 \end{bmatrix}$
$\mathbf{w_2}: \begin{bmatrix} 0,0,\ldots,0 \\ weights\_2 \\ 0,0,\ldots,0 \end{bmatrix}$
$\mathbf{w_3}: \begin{bmatrix} 0,0,\ldots,0 \\ 0,0,\ldots,0 \\ weights\_3 \end{bmatrix}$
$\mathbf{w_1}: [weights\_1]$
$\mathbf{w_2}: [weights\_2]$
$\mathbf{w_3}: [weights\_3]$

Figure 14: The transformation of $\mathbf{w}_i$ in $\boldsymbol{W}$ when all labels are distinct. In the example, there are 3 sequence labelling tasks, and it is assumed that each task has exactly one label different from those of other tasks. Therefore, there are 3 distinct labels in total, and each $\mathbf{w}_i$ is expanded by appending 0.

With weight tensor alignment and the model structure in Section 5.2, the connection between MRN and Multi-Task sequence labelling is built to make the Transformer-based models estimate Multi-Linear relationships to perform better. Whereas, there are still some potential constraints which will be mentioned in Chapter 8.

Weight tensor alignment can be simply implemented with the Pytorch framework where “torch.cat” and “torch.zeros” functions are used for concatenating tensors and appending 0, respectively. Regarding the detailed implementation code, it will be provided in A.5 (like the function “weights_cat_conll2003”).

5.4 MRN Regularization and Correlation Matrices

Except for the weight tensor alignment, the calculation of the regularization term L_{reg} follows the equations in Section 3.3.2. The AdamW optimizer optimizes all the trainable parameters in the composite model first, and after every certain number of batches of samples are processed, all the covariance matrices are updated according to the equation (24), (25), and (26).

Next, since the new covariance matrices are gained after the training stage, the correlation matrices can be derived. Note that $\mathbf{\Omega}_1 \in \mathbb{R}^{D_1 \times D_1}$, $\mathbf{\Omega}_2 \in \mathbb{R}^{D_2 \times D_2}$, and $\mathbf{\Omega}_3 \in \mathbb{R}^{D_3 \times D_3}$, where D_1 is the number of features from the former shared layer, D_2 is the total number of distinct labels across all sequence labelling tasks, and D_3 is the number of sequence labelling tasks. The correlation matrix $\mathbf{R}$ can be solved by the equation (27):

$$\mathbf{R} = \left(\mathbf{C}_{\mathbf{diag}}\right)^{-1}\mathbf{C}\left(\mathbf{C}_{\mathbf{diag}}\right)^{-1}, \tag{27}$$

Where $\mathbf{C}$ is any covariance matrix acquired after training, and $\mathbf{C}_{\mathbf{diag}}$ is the diagonal matrix of $\mathbf{C}$. Since $\mathbf{C}_{\mathbf{diag}}$ is a diagonal matrix, $\left(\mathbf{C}_{\mathbf{diag}}\right)^{-1}$ equals $(\left(\mathbf{C}_{\mathbf{diag}}\right)^{-1})^{\mathrm{T}}$. With given $\mathbf{R}$, all pairwise correlations can be inspected, and the task relationships learnt between all sequence labelling tasks can be known when setting $\mathbf{C}$ to $\mathbf{\Omega}_3$.

The implementation of L_{reg} and correlation matrices is strongly related to tensor multiplication, transposition, and decomposition. These operations can be implemented with built-in functions (“torch.mm”, “torch.t”, and “torch.svd”, etc.) in Pytorch. The AdamW optimizer can also be implemented with “torch.optim.AdamW”. The code for calculating L_{reg} and updating covariance matrices will be shown in A.5 (the functions “loss_reg” and “covariance_mat_update”), and the code for calculating correlation matrices will be given in the last block in A.2 (the function “calculate_correlation”).

6 Results, Analysis and Evaluation

The chapter will analyse and evaluate the results gained from the experiment to justify that the aims and objectives are accomplished. The exact parameter settings together with evaluation criteria will be given.

6.1 Parameter Settings

The architecture of the composite models follows their structure in Section 5.2. As mentioned before, for the bottom of the shared layers, pre-trained BERT$_{base}$ and RoBERTa$_{base}$ models are downloaded from Huggingface's transformers, and BERT$_{base}$ is the “cased” version.

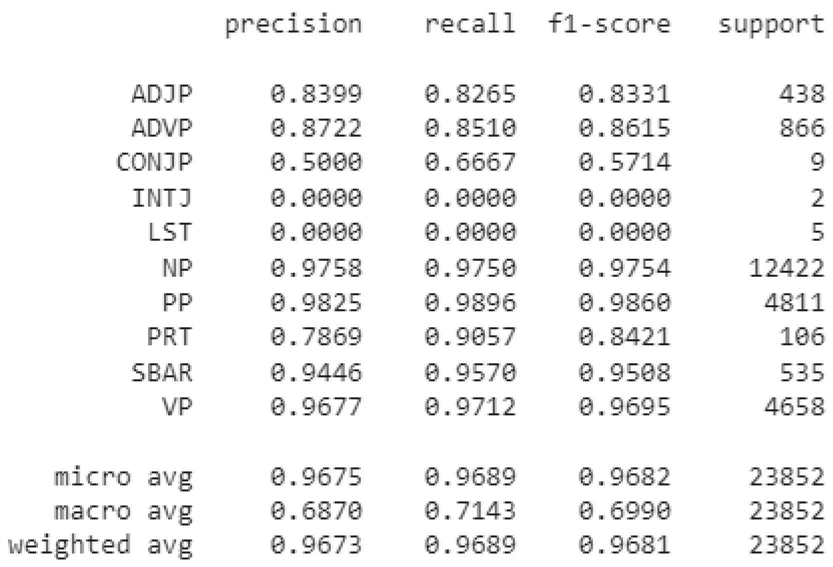
Both of the layers were pre-trained on the datasets mentioned in their original papers [7], [18]. The configurations of the inner layers of BERT$_{base}$ and RoBERTa$_{base}$ are the same as those in Sections 2.5 and 2.6. For the BiLSTM layer, the hidden size is chosen as 768 for both forward and backward LSTM. The input dimension of each linear layer depends on the output dimension of the formerly shared layer. For the BERT$_{base}$- BiLSTM-Linear model, the input dimension of each of its linear layers is 1536, and for other models, the input dimension of their linear layers is 768.

In the project, the maximum input sequence length is 256, and all the batch sizes for training, testing and validation are set to 16 for both of the datasets. As for the settings in the training stage, all the training hyper-parameters for all composite models are the same. The models are fine-tuned for at most 5 epochs on all the datasets with the learning rate of 5e-5 decaying linearly to 0. The settings for AdamW are: (β_1, β_2)=(0.9, 0.999), ϵ=1e-8, weight decay ratio=1e-2. When the models are trained under MRN, the values of λ are chosen from $\{1, 0.5, 0.05\}$, and ε is fixed at 1e-5 to get the best results. The frequency for updating covariance matrices is set to 10, which means that the matrices are updated after every 10 batches of training samples are processed.

6.2 Evaluation Metrics

As regards the testing stage, the python module “seqeval” [44] is used for evaluation, and it is compatible with the original scoring script “conlleval” referred to in [45]. According to the prediction methodology in Section 5.2, the raw output tensors of the composite model are transformed into the labels for each sequence labelling task. Nevertheless, because some true labels are inserted “-100” in certain positions, they have to be removed. This can be simply done by finding where the “-100” labels are in the ground-truth labels, and then all the labels with these positions in the true labels and the predicted labels are deleted.

After the deletion, the functions in the “seqeval” module such as “classification_report” can be used to calculate desired scores for evaluation. Figure 15 illustrates the chunking results of CoNLL-2000 under MRN with RoBERTa$_{base}$-Linear. The results are obtained by the “classification_report” function which specifies each evaluation score, and the micro average is used to follow the evaluation paradigms of the datasets.



```
              precision    recall  f1-score   support

        ADJP     0.8399    0.8265    0.8331       438
        ADVP     0.8722    0.8510    0.8615       866
       CONJP     0.5000    0.6667    0.5714         9
        INTJ     0.0000    0.0000    0.0000         2
         LST     0.0000    0.0000    0.0000         5
          NP     0.9758    0.9750    0.9754     12422
          PP     0.9825    0.9896    0.9860      4811
         PRT     0.7869    0.9057    0.8421       106
        SBAR     0.9446    0.9570    0.9508       535
          VP     0.9677    0.9712    0.9695      4658

   micro avg     0.9675    0.9689    0.9682     23852
   macro avg     0.6870    0.7143    0.6990     23852
weighted avg     0.9673    0.9689    0.9681     23852
```

Figure 15: The example chunking results of CoNLL-2000 under MRN with a RoBERTa$_{base}$-Linear model.

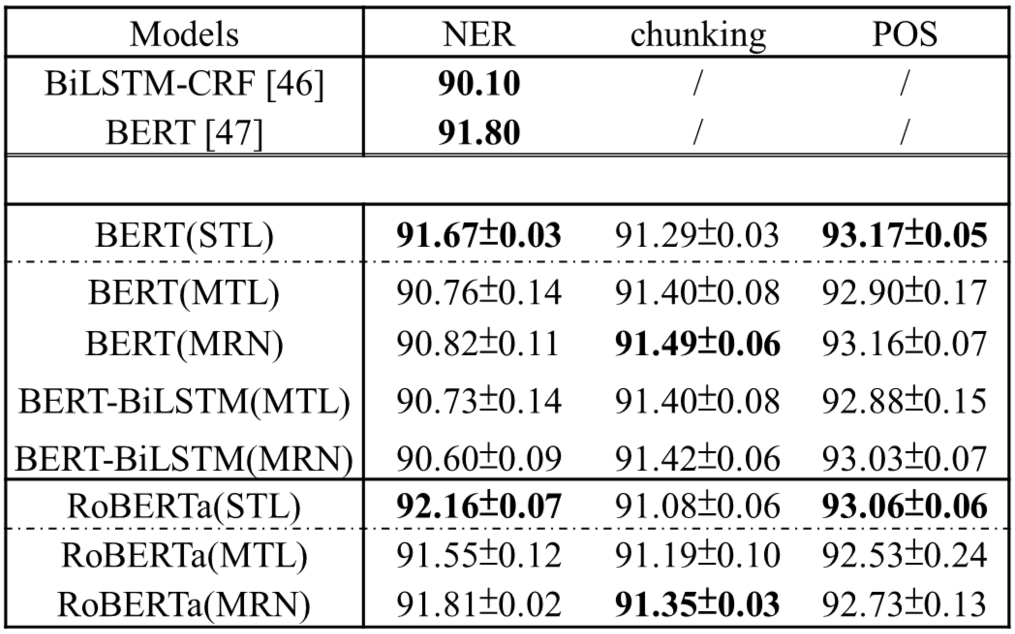
6.3 Results with Analysis

The results are acquired based on the implementation of the proposed methodology, and they are summarized in 3 tables. Each element in a table shows the mean followed by the standard deviation calculated from the F1-scores from 5 random starts. Table 1 lists the scores gained from only using the CoNLL-2003 dataset, and Table 2 gives the implementation results acquired from only using the CoNLL-2000 dataset. In the two implementations, since the output space of each task is distinct, the alignment of weight tensors follows the method in Figure 14. To show that MRN can indeed learn the relations of tasks, features, and labels when there are some overlapping labels between sequence labelling tasks, Table 3 shows the experiment results gained by training the MTL and MRN models with the POS tagging tasks of CoNLL-2000 and CoNLL-2003. The scores are acquired using 5% and 10% of the samples to see if MRN can perform better than the pure MTL method. Besides, in this scenario, the alignment of weight tensors follows the method in Figure 13.

The F1-score of the BiLSTM-CRF model is referenced from [46], and researchers in [47] also use Huggingface’s transformers the same as those in this project. As Table 1 shows, STL outperforms MTL and MRN on NER and POS tasks, and MRN is better than its counterparts on text chunking. The performance on chunking is better when the models are fine-tuned under MTL or MRN. Apart from these, for all the 3 different composite models, MRN outperforms pure MTL on all tasks except for the NER task using BERT-BiLSTM.

The score of BERT(STL) is 91.67% which is close enough to that in [47], and it could be seen that the results in the project are aligned with those in the reference papers. The obtained results imply that MTL and MRN may not able to win compared to STL in terms of F1-scores. However, as referred to before, the training stage of the STL models is not resource-efficient since the number of models required to be fine-tuned increases linearly with the number of tasks. Furthermore, an advantage of Multi-Task models lies in that some tasks may benefit from shared knowledge between tasks. For instance, the chunking task in CoNLL-2003 benefits from MTL and MRN, and the scores of MRN models are even higher than those of MTL ones. In addition, the Multi-Task models still fail to prevent negative transfer among tasks. Whereas, the results in the table imply that MRN can slightly alleviate this problem since MRN almost outperforms MTL on all tasks. This can be seen that MRN is better than pure MTL because MRN explicitly models the relations.

Among all shared encoder layers, BERT is better on chunking and POS tagging tasks, and RoBERTa outperforms others on the NER task. Therefore, it is hard to say BERT or RoBERTa is more powerful by simply testing them with these 3 sequence labelling tasks. However, Table 1 shows that adding BiLSTM layers does not improve the performance of BERT. This may be due to 2 reasons. Firstly, adding BiLSTM layers may lead to the problem of over-fitting since the number of parameters increases. Secondly, the pre-trained models have already been more powerful than BiLSTM and stored sufficient contextual information.



Models	NER	chunking	POS
BiLSTM-CRF [46]	**90.10**	/	/
BERT [47]	**91.80**	/	/
BERT(STL)	**91.67±0.03**	91.29±0.03	**93.17±0.05**
BERT(MTL)	90.76±0.14	91.40±0.08	92.90±0.17
BERT(MRN)	90.82±0.11	**91.49±0.06**	93.16±0.07
BERT-BiLSTM(MTL)	90.73±0.14	91.40±0.08	92.88±0.15
BERT-BiLSTM(MRN)	90.60±0.09	91.42±0.06	93.03±0.07
RoBERTa(STL)	**92.16±0.07**	91.08±0.06	**93.06±0.06**
RoBERTa(MTL)	91.55±0.12	91.19±0.10	92.53±0.24
RoBERTa(MRN)	91.81±0.02	**91.35±0.03**	92.73±0.13

Table 1: The F1-scores on CoNLL-2003 under different learning methods with the 3 composite models. Note that the "-Linear" notations are ignored in the table only for simplicity. For example, BERT(STL) is equal to BERT$_{base}$-Linear(STL), and BERT(MRN) is identical to BERT$_{base}$-Linear(MRN).

In Table 2, The F1-score gained by BiLSTM-CRF is also referenced from [46], and [48] uses Huggingface's transformers and Huggingface's datasets exactly identical to those in this project. The F1-scores of the Flair and the GCDT/BERT models are referenced from [49] for further comparisons. GCDT/BERT is listed to see whether the models trained in this project can perform better than such an advanced model. The best result gained in CoNLL-2000 is the score on text chunking using the RoBERTa$_{base}$-Linear MRN model (96.85%) which is slightly higher than those in [49].

Similar to the results in the former table, MTL and MRN cannot outperform STL most of the time, and MRN is better than MTL under all circumstances. This may imply that MRN is more powerful than MTL since MRN can regularize the models to learn better Multi-Linear relationships.

As for the shared encoder models, the observations are similar to those for Table 1 except that RoBERTa is greater on chunking and BERT performs better on POS tagging.

Models	chunking	POS
BiLSTM-CRF [46]	**94.46**	/
BERT [48]	**95.92**	/
RoBERTa [48]	**96.35**	/
Flair [49]	**96.72**	/
GCDT w/ BERT [49]	**96.81**	/
BERT(STL)	**96.63±0.04**	**98.25±0.11**
BERT(MTL)	96.31±0.10	97.94±0.07
BERT(MRN)	96.47±0.07	98.02±0.05
BERT-BiLSTM(MTL)	96.29±0.11	97.94±0.03
BERT-BiLSTM(MRN)	96.44±0.09	97.99±0.07
RoBERTa(STL)	96.71±0.05	**98.17±0.09**
RoBERTa(MTL)	96.76±0.04	97.75±0.09
RoBERTa(MRN)	**96.85±0.04**	97.86±0.07

Table 2: The F1-scores on CoNLL-2000 under different learning methods with BERT$_{base}$-Linear and RoBERTa$_{base}$-Linear.

The figure in the above 2 tables is obtained under the situation that the output spaces of all sequence labelling tasks are completely different. Accordingly, it is necessary to verify whether the modified MRN can perform well when some labels are the same between tasks. Table 3 shows the results acquired by using 5% and 10% of the combination of POS tagging samples from both of the datasets using BERT$_{base}$-Linear under MTL and MRN. Namely, when the proportion is set to 5%, 5% of the samples in CoNLL-2003 and 5% of samples in CoNLL-2000 are randomly selected to form a new dataset. Furthermore, the POS tagging task in CoNLL-2003 has 3 more labels different from those in CoNLL-2000, and there are 44 identical labels among the two tasks. Overall, there are totally 2 sequence labelling tasks which are POS tagging tasks of CoNLL-2003 and CoNLL-2000, and all the kinds of POS tags in CoNLL-2000 are included in CoNLL-2003.

As the table shows, MRN outperforms MTL even if there are few samples for training. It is obvious to see that modelling relationships is significant, and MRN can successfully regularize the models to obtain better results.

Models	POS 5% (CoNLL-2003)	POS 5% (CoNLL-2000)	POS 10% (CoNLL-2003)	POS 10% (CoNLL-2000)
BERT(MTL)	90.51±0.17	95.03±0.10	91.48±0.10	96.49±0.07
BERT(MRN)	**90.61±0.11**	**95.12±0.13**	**91.60±0.07**	**96.57±0.08**

Table 3: The POS tagging F1-scores on the combination of the two datasets under MTL or MRN with BERT$_{base}$-Linear.

Moreover, to visualize the Multi-Linear relationships that MRN models learn, Figure 16 illustrates the task relations learnt by the models, and Figure 17 displays the label and feature relationships. The colours red and blue denote the positive and negative correlations, respectively. The white colour means zero correlations.

It is indicated in the figures that MRN can guide the models to learn better relations. For example, Figure 16 shows that the tasks are positively correlated since 44 labels are the same in the two tasks. Apart from the task relationships, MRN also make models learn pairwise relations across all types of labels and features.

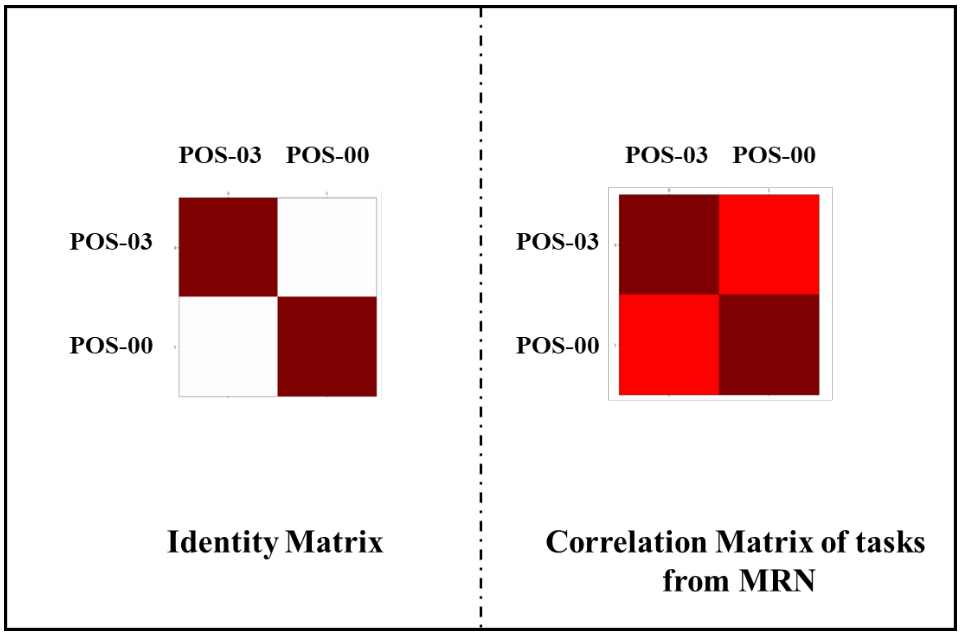


Figure 16: The task correlation matrix learnt by MRN models compared to an identity matrix. POS-03 is the POS tagging task in CoNLL-2003 and POS-00 is that in CoNLL-2000.

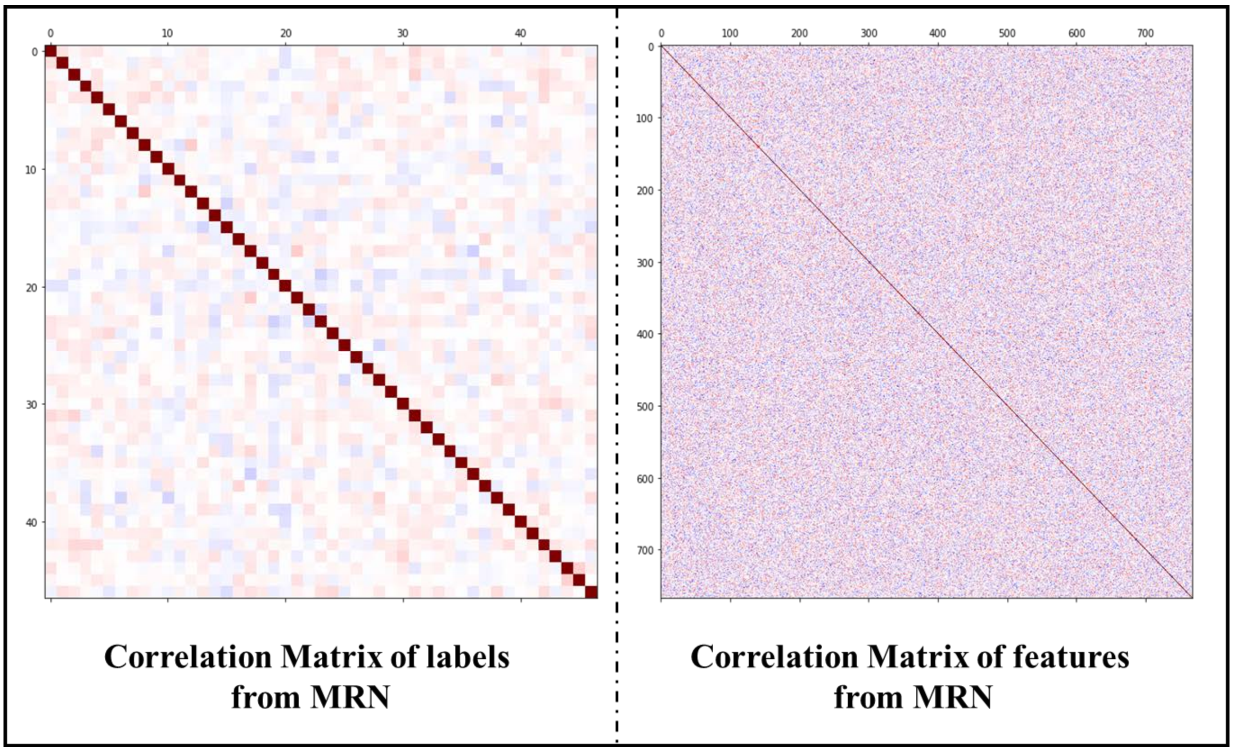


Figure 17: The label and the feature correlation matrices learnt by BERT$_{base}$-Linear MRN models. There are 47 POS labels across all tasks, and 44 (label index starts from 0 to 43) of them are shared by the two tasks. There are 768 features, and this equals the number of output features from the former BERT$_{base}$ layer.

As regards the achievement, the framework is successfully implemented based on the objectives followed by requirements and specifications to accomplish the aims. Overall, it could be summarized that the framework can indeed estimate the relations and outperform its MTL counterpart most of the time, and the framework is better than its STL counterpart in terms of scalability for multiple tasks. Apart from these, the performance of different Transformer encoders differs. Namely, their performance is strongly related to the choices of the datasets. A model can perform better on a task in a dataset but achieve worse outcomes on the same task on another dataset, so it cannot be concluded which encoder (BERT or RoBERTa) is more powerful for Multi-Task sequence labelling.

7 Ethical, Social, Legal and Professional Issues

In the chapter, the various issues related to the whole project will be considered. The concerns of ethics and societies will be discussed, and potential issues about legality and profession will be carefully considered.

7.1 Ethical and Social Issues

In the work of this project, a possible issue may arise when the models are used for unethical sequence labelling tasks. For example, detecting racial entities from a biased dataset could produce models with racial discrimination. If these fine-tuned models were used for further applications or abuses, there would be a social impact. Therefore, it is claimed that all the work in the project is not designed or implemented for any potential abuse.

7.2 Legal and Professional Concerns

When it comes to the concerns of legality and profession, the principles (Code of Good Practice and Code of Conduct) from the British Computer Society [50] are attentively followed throughout this project. Meanwhile, I am aware of the guidance (Rule of Conduct) from The Institution of Engineering and Technology [51].

The pre-trained models, datasets, relevant code, and libraries referred to are all open source. The specifications for implementation were listed, and the mentioned background theories with their original papers were referenced properly. Accordingly, it is important to clarify that I do not possess or claim ownership of any previous study or work from others.

8 Conclusion

In this project, the modified MRN framework with a shared Transformer encoder layer for sequence labelling is successfully implemented by aligning weight tensors of all linear layers, and this further generalizes MRN to sequence labelling tasks, no matter how many labels are shared among tasks. As the results show, MRN can improve the performance on the benchmark datasets in comparison to MTL, and this may indicate that MRN can indeed alleviate the potential transferring problems occurring in MTL. Besides, MRN regularizes the models with the task, the label, and the feature relations to learn better with few samples compared to pure MTL. The MRN and MTL methods still cannot outperform STL in general, but MRN and MTL enhance the efficiency in the training stage by sharing some layers in a model. Furthermore, the relations can be visualized to verify that MRN can indeed estimate the explicit relationships.

Regarding the comparison of BERT and RoBERTa in the context of sequence labelling, it is hard to say which one outperforms the other. Different tasks or datasets result in various situations. A model may perform better on a certain task of a dataset but perform worse than the other on other datasets even if the task is the same. However, it could be concluded that adding a shared BiLSTM layer to a BERT-like model does not help the improvement of the model. A BiLSTM layer may hurt the performance since the information contained in a pre-trained Transformer-based model has already been sufficient, and the BiLSTM layer may also cause the over-fitting problem due to the growth in the number of parameters.

Although the way of applying MRN sequence labelling is proposed to improve the performance on the benchmark datasets, there are still some limitations. When multiple labels in a task correspond to one label in another task, the current proposed tensor aligning method is not applicable. The proposed label-combining method can only deal with the one-to-one label correspondence, and the label has to be exactly the same. Otherwise, the proposed tensor aligning method treats such related labels independent of others, and this may lead to decreases in the quality of relations that a model learns. For instance, it is assumed that the labels “I-NP” and “B-NP” in a task correspond to the label “NP” in another task, and then the method fails to model such correspondence because it can only deal with the situation like aligning “I-NP” in a task and “I-NP” in another task. Therefore, future work may lie in how to cope with such situations so that MRN can make models learn more accurate relations.

9 REFERENCES

[1] D. Nadeau and S. Sekine, “A survey of named entity recognition and classification,” *Lingvisticae Investigationes*, vol. 30, no. 1, pp. 3–26, 2007.

[2] R. Caruana, “Multitask learning,” *Machine learning*, vol. 28, no. 1, pp. 41–75, 1997.

[3] Y. Zhang and Q. Yang, “A survey on multi-task learning,” *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[4] P. Cao, Y. Chen, K. Liu, J. Zhao, and S. Liu, “Adversarial transfer learning for Chinese named entity recognition with self-attention mechanism,” in *Proceedings of the 2018 conference on empirical methods in natural language processing*, 2018, pp. 182–192.

[5] Y. Zhang and D.-Y. Yeung, “A regularization approach to learning task relationships in multitask learning,” *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 8, no. 3, pp. 1–31, 2014.

[6] Y. Zhang and D.-Y. Yeung, “Multi-task boosting by exploiting task relationships,” in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2012, pp. 697–710.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[8] M. Rei and H. Yannakoudakis, “Compositional sequence labeling models for error detection in learner writing,” *arXiv preprint arXiv:1607.06153*, 2016.

[9] M. Rei and H. Yannakoudakis, “Auxiliary objectives for neural error detection models,” *arXiv preprint arXiv:1707.05227*, 2017.

[10] X. Ma and E. Hovy, “End-to-end sequence labeling via bi-directional lstm-cnns-crf,” arXiv preprint arXiv:1603.01354, 2016.

[11] S. Changpinyo, H. Hu, and F. Sha, “Multi-task learning for sequence tagging: An empirical study,” *arXiv preprint arXiv:1808.04151*, 2018.

[12] P. Lu, T. Bai, and P. Langlais, “Sc-lstm: Learning task-specific representations in multi-task learning for sequence labeling,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 2396–2406.

[13] A. Vaswani et al., “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

[14] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, “Transformers are rnns: Fast autoregressive transformers with linear attention,” in *International Conference on Machine Learning*, 2020, pp. 5156–5165.
[15] X. Liu, P. He, W. Chen, and J. Gao, “Multi-task deep neural networks for natural language understanding,” *arXiv preprint arXiv:1901.11504*, 2019.
[16] T. Bao, N. Ren, R. Luo, B. Wang, G. Shen, and T. Guo, “A BERT-Based Hybrid Short Text Classification Model Incorporating CNN and Attention-Based BiGRU,” *Journal of Organizational and End User Computing (JOEUC)*, vol. 33, no. 6, pp. 1–21, 2021.
[17] Y. Han, Y. Yan, Y. Han, R. Chao, and H. Zan, “Chinese Grammatical Error Diagnosis Based on RoBERTa-BiLSTM-CRF Model,” in *Proceedings of the 6th Workshop on Natural Language Processing Techniques for Educational Applications*, 2020, pp. 97–101.
[18] Y. Liu et al., “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.
[19] Z. Dai, X. Wang, P. Ni, Y. Li, G. Li, and X. Bai, “Named entity recognition using BERT BiLSTM CRF for Chinese electronic health records,” in *2019 12th international congress on image and signal processing, biomedical engineering and informatics (cisp-bmei)*, 2019, pp. 1–5.
[20] P. Porkaew, P. Boonkwan, and T. Supnithi, “HoogBERTa: Multi-task Sequence Labeling using Thai Pretrained Language Representation,” in *2021 16th International Joint Symposium on Artificial Intelligence and Natural Language Processing (iSAI-NLP)*, 2021, pp. 1 – 6.
[21] Y. Wang, Y. Li, Z. Zhu, H. Tong, and Y. Huang, “Adversarial learning for multi-task sequence labeling with attention mechanism,” *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 28, pp. 2476–2488, 2020.
[22] X. Chen, Z. Shi, X. Qiu, and X. Huang, “Adversarial multi-criteria learning for chinese word segmentation,” *arXiv preprint arXiv:1704.07556*, 2017.
[23] C. Fan, C. Yuan, L. Gui, Y. Zhang, and R. Xu, “Multi-task sequence tagging for emotion-cause pair extraction via tag distribution refinement,” *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 29, pp. 2339–2350, 2021.
[24] Y. Xue, X. Liao, L. Carin, and B. Krishnapuram, “Multi-Task Learning for Classification with Dirichlet Process Priors.,” *Journal of Machine Learning Research*, vol. 8, no. 1, 2007.
[25] E. F. Sang and F. De Meulder, “Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition,” *arXiv preprint cs/0306050*, 2003.
[26] L. Wang, *Support vector machines: theory and applications*, vol. 177. Springer Science & Business Media, 2005.
[27] Y. Lin, S. Yang, V. Stoyanov, and H. Ji, “A multi-lingual multi-task architecture for low-resource sequence labeling,” in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018, pp. 799–809.
[28] A. K. Gupta and D. K. Nagar, *Matrix variate distributions*. Chapman, 2018.
[29] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[30] A. Radford et al., “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
[31] S. J. Pan, “Transfer learning,” *Learning*, vol. 21, pp. 1–2, 2020.
[32] M. Long, Z. Cao, J. Wang, and P. S. Yu, “Learning multiple tasks with multilinear relationship networks,” *Advances in neural information processing systems*, vol. 30, 2017.
[33] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” *arXiv preprint arXiv:1711.05101*, 2017.
[34] E. Bisong, *Building machine learning and deep learning models on Google cloud platform: A comprehensive guide for beginners*. Apress, 2019.
[35] A. Paszke et al., “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.

[36] E. F. Sang and S. Buchholz, "Introduction to the CoNLL-2000 shared task: Chunking," *arXiv preprint cs/0009008*, 2000.
[37] Q. Lhoest et al., "Datasets: A community library for natural language processing," *arXiv preprint arXiv:2109.02846*, 2021.
[38] L. V. Lita, A. Ittycheriah, S. Roukos, and N. Kambhatla, "Truecasing," in *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, 2003, pp. 152–159.
[39] T. Wolf et al., "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.
[40] Y. Zhu et al., "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 19–27.
[41] A. Gokaslan and V. Cohen, *OpenWebText Corpus*. 2019. [Online]. Available: http://Skylion007.github.io/OpenWebTextCorpus [Accessed: July 01, 2022].
[42] T. H. Trinh and Q. V. Le, "A simple method for commonsense reasoning," *arXiv preprint arXiv:1806.02847*, 2018.
[43] S. Nagel, *Cc-news*. 2016. [Online]. Available: https://commoncrawl.org/2016/10/news-dataset-available/ [Accessed: June 06, 2022].
[44] H. Nakayama, *seqeval: A Python framework for sequence labeling evaluation*. 2018. [Online]. Available: https://github.com/chakki-works/seqeval [Accessed: June 05, 2022].
[45] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of machine learning research*, vol. 12, no. ARTICLE, pp. 2493–2537, 2011.
[46] Z. Huang, W. Xu, and K. Yu, "Bidirectional LSTM-CRF models for sequence tagging," *arXiv preprint arXiv:1508.01991*, 2015.
[47] B. Li, "Named entity recognition in the style of object detection," *arXiv preprint arXiv:2101.11122*, 2021.
[48] C. Poth, J. Pfeiffer, A. Rücklé, and I. Gurevych, "What to pre-train on? efficient intermediate task selection," *arXiv preprint arXiv:2104.08247*, 2021.
[49] L Y. Li, L. Liu, and S. Shi, "Segmenting natural language sentences via lexical unit analysis," *arXiv preprint arXiv:2012.05418*, 2020.
[50] British Computing Society, *BCS Code of Conduct*. 2022. [Online]. Available: https://www.bcs.org/membership-and-registrations/become-a-member/bcs-code-of-conduct/ [Accessed: July 08, 2022].
[51] The Institution of Engineering and Technology. *Rules of Conduct*. 2022. [Online]. Available: https://www.theiet.org/about/governance/rules-of-conduct/ [Accessed: July 08, 2022].

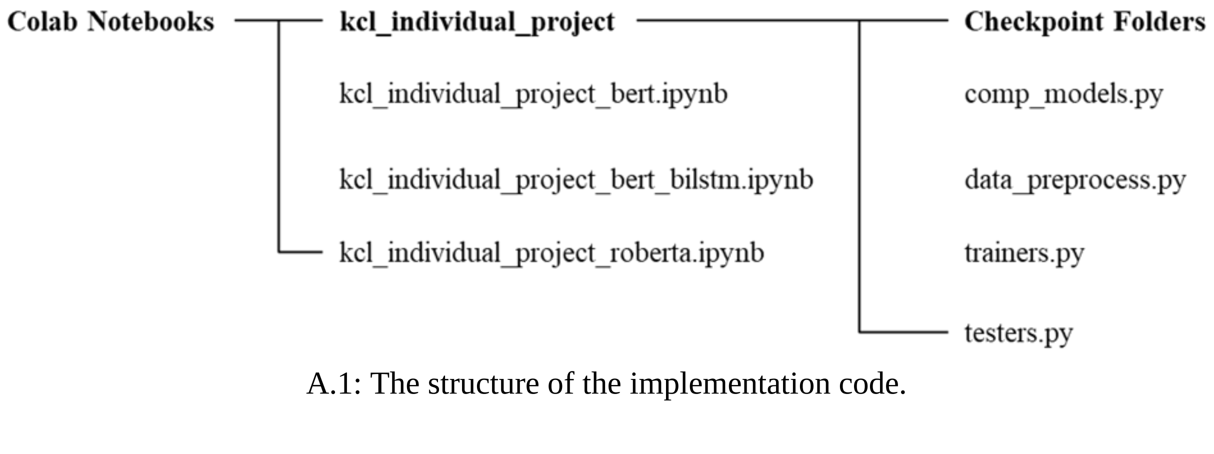
10 APPENDICES

In this chapter, the code for implementation will be provided, and the structure of the code with the directory hierarchy will also be given.

10.1 Appendix A: Implementation Code

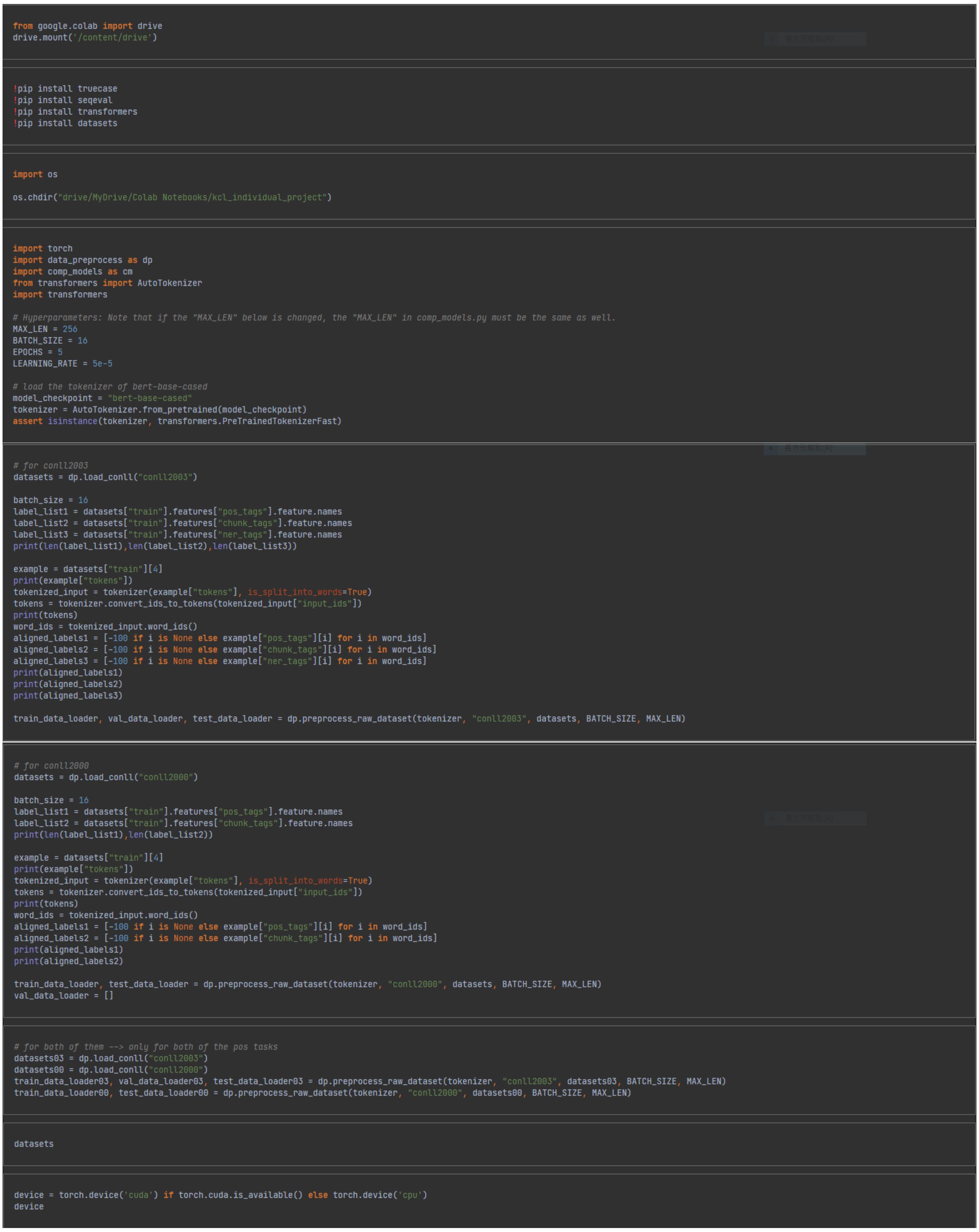
A.1 shows the structure of the code. The words in bold denote folders, and the rest are files. Note that the user should build the “Checkpoint Folders” to specify where to save the models. There are 3 “.ipynb” files which guide the user to train a composite model under a learning paradigm and test the model to show its F1-scores. The words follow “kcl_individual_project_” means the Transformer-based model to be chosen. For example, “kcl_individual_project_bert.ipynb” (A.2) indicates that the user wants to use a BERT-Linear model. Basically, the user can directly execute one of these 3 files after downloading and building the “Checkpoint Folders”.

The “comp_models.py” file (A.3) specifies the structure of all composite models, and “data_preprocess.py” (A.4) is the file for data pre-processing and loading. “trainers.py” (A.5) defines how the models are trained, and “testers.py” (A.6) specifies how the models are evaluated.



Colab Notebooks
kcl_individual_project
kcl_individual_project_bert.ipynb
kcl_individual_project_bert_bilstm.ipynb
kcl_individual_project_roberta.ipynb
Checkpoint Folders
comp_models.py
data_preprocess.py
trainers.py
testers.py

A.1: The structure of the implementation code.



```
from google.colab import drive
drive.mount('/content/drive')
```

```
!pip install truecase
!pip install seqeval
!pip install transformers
!pip install datasets
```

```
import os

os.chdir("drive/MyDrive/Colab Notebooks/kcl_individual_project")
```

```
import torch
import data_preprocess as dp
import comp_models as cm
from transformers import AutoTokenizer
import transformers

# Hyperparameters: Note that if the "MAX_LEN" below is changed, the "MAX_LEN" in comp_models.py must be the same as well.
MAX_LEN = 256
BATCH_SIZE = 16
EPOCHS = 5
LEARNING_RATE = 5e-5

# load the tokenizer of bert-base-cased
model_checkpoint = "bert-base-cased"
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
assert isinstance(tokenizer, transformers.PreTrainedTokenizerFast)
```

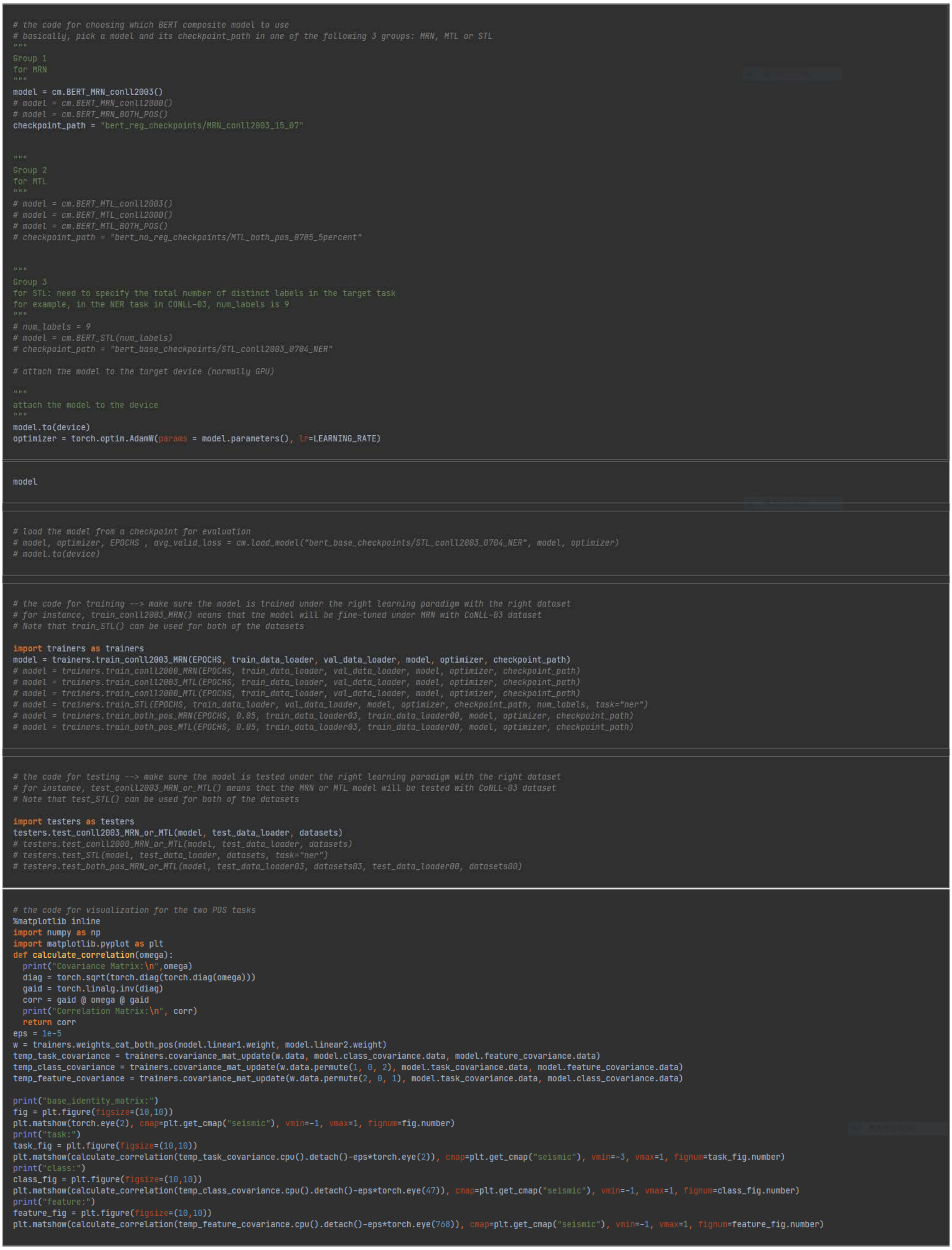
```
# for conll2003
datasets = dp.load_conll("conll2003")

batch_size = 16
label_list1 = datasets["train"].features["pos_tags"].feature.names
label_list2 = datasets["train"].features["chunk_tags"].feature.names
label_list3 = datasets["train"].features["ner_tags"].feature.names
print(len(label_list1),len(label_list2),len(label_list3))

example = datasets["train"][4]
print(example["tokens"])
tokenized_input = tokenizer(example["tokens"], is_split_into_words=True)
tokens = tokenizer.convert_ids_to_tokens(tokenized_input["input_ids"])
print(tokens)
word_ids = tokenized_input.word_ids()
aligned_labels1 = [-100 if i is None else example["pos_tags"][i] for i in word_ids]
aligned_labels2 = [-100 if i is None else example["chunk_tags"][i] for i in word_ids]
aligned_labels3 = [-100 if i is None else example["ner_tags"][i] for i in word_ids]
print(aligned_labels1)
print(aligned_labels2)
print(aligned_labels3)

train_data_loader, val_data_loader, test_data_loader = dp.preprocess_raw_dataset(tokenizer, "conll2003", datasets, BATCH_SIZE, MAX_LEN)
```

```
# for conll2000
datasets = dp.load_conll("conll2000")

batch_size = 16
label_list1 = datasets["train"].features["pos_tags"].feature.names
label_list2 = datasets["train"].features["chunk_tags"].feature.names
print(len(label_list1),len(label_list2))

example = datasets["train"][4]
print(example["tokens"])
tokenized_input = tokenizer(example["tokens"], is_split_into_words=True)
tokens = tokenizer.convert_ids_to_tokens(tokenized_input["input_ids"])
print(tokens)
word_ids = tokenized_input.word_ids()
aligned_labels1 = [-100 if i is None else example["pos_tags"][i] for i in word_ids]
aligned_labels2 = [-100 if i is None else example["chunk_tags"][i] for i in word_ids]
print(aligned_labels1)
print(aligned_labels2)

train_data_loader, test_data_loader = dp.preprocess_raw_dataset(tokenizer, "conll2000", datasets, BATCH_SIZE, MAX_LEN)
val_data_loader = []
```

```
# for both of them --> only for both of the pos tasks
datasets03 = dp.load_conll("conll2003")
datasets00 = dp.load_conll("conll2000")
train_data_loader03, val_data_loader03, test_data_loader03 = dp.preprocess_raw_dataset(tokenizer, "conll2003", datasets03, BATCH_SIZE, MAX_LEN)
train_data_loader00, test_data_loader00 = dp.preprocess_raw_dataset(tokenizer, "conll2000", datasets00, BATCH_SIZE, MAX_LEN)
```

```
datasets
```

```
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
device
```



```python
# the code for choosing which BERT composite model to use
# basically, pick a model and its checkpoint_path in one of the following 3 groups: MRN, MTL or STL
"""
Group 1
for MRN
"""
model = cm.BERT_MRN_conll2003()
# model = cm.BERT_MRN_conll2000()
# model = cm.BERT_MRN_BOTH_POS()
checkpoint_path = "bert_reg_checkpoints/MRN_conll2003_15_07"


"""
Group 2
for MTL
"""
# model = cm.BERT_MTL_conll2003()
# model = cm.BERT_MTL_conll2000()
# model = cm.BERT_MTL_BOTH_POS()
# checkpoint_path = "bert_no_reg_checkpoints/MTL_both_pos_0705_5percent"


"""
Group 3
for STL: need to specify the total number of distinct labels in the target task
for example, in the NER task in CONLL-03, num_labels is 9
"""
# num_labels = 9
# model = cm.BERT_STL(num_labels)
# checkpoint_path = "bert_base_checkpoints/STL_conll2003_0704_NER"

# attach the model to the target device (normally GPU)

"""
attach the model to the device
"""
model.to(device)
optimizer = torch.optim.AdamW(params = model.parameters(), lr=LEARNING_RATE)
```

```python
model
```

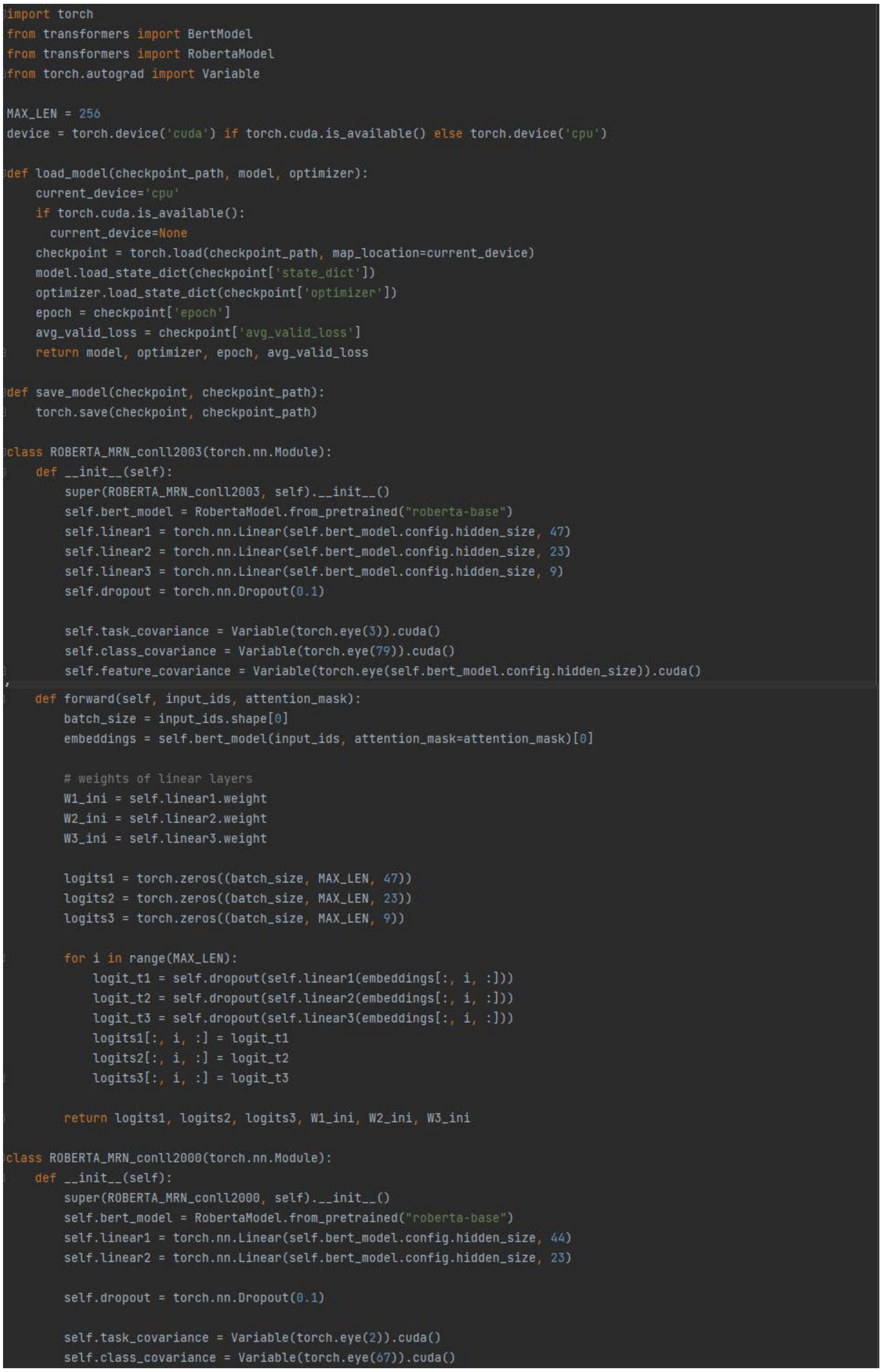
```python
# load the model from a checkpoint for evaluation
# model, optimizer, EPOCHS , avg_valid_loss = cm.load_model("bert_base_checkpoints/STL_conll2003_0704_NER", model, optimizer)
# model.to(device)
```

```python
# the code for training --> make sure the model is trained under the right learning paradigm with the right dataset
# for instance, train_conll2003_MRN() means that the model will be fine-tuned under MRN with CoNLL-03 dataset
# Note that train_STL() can be used for both of the datasets

import trainers as trainers
model = trainers.train_conll2003_MRN(EPOCHS, train_data_loader, val_data_loader, model, optimizer, checkpoint_path)
# model = trainers.train_conll2000_MRN(EPOCHS, train_data_loader, val_data_loader, model, optimizer, checkpoint_path)
# model = trainers.train_conll2003_MTL(EPOCHS, train_data_loader, val_data_loader, model, optimizer, checkpoint_path)
# model = trainers.train_conll2000_MTL(EPOCHS, train_data_loader, val_data_loader, model, optimizer, checkpoint_path)
# model = trainers.train_STL(EPOCHS, train_data_loader, val_data_loader, model, optimizer, checkpoint_path, num_labels, task="ner")
# model = trainers.train_both_pos_MRN(EPOCHS, 0.05, train_data_loader03, train_data_loader00, model, optimizer, checkpoint_path)
# model = trainers.train_both_pos_MTL(EPOCHS, 0.05, train_data_loader03, train_data_loader00, model, optimizer, checkpoint_path)
```

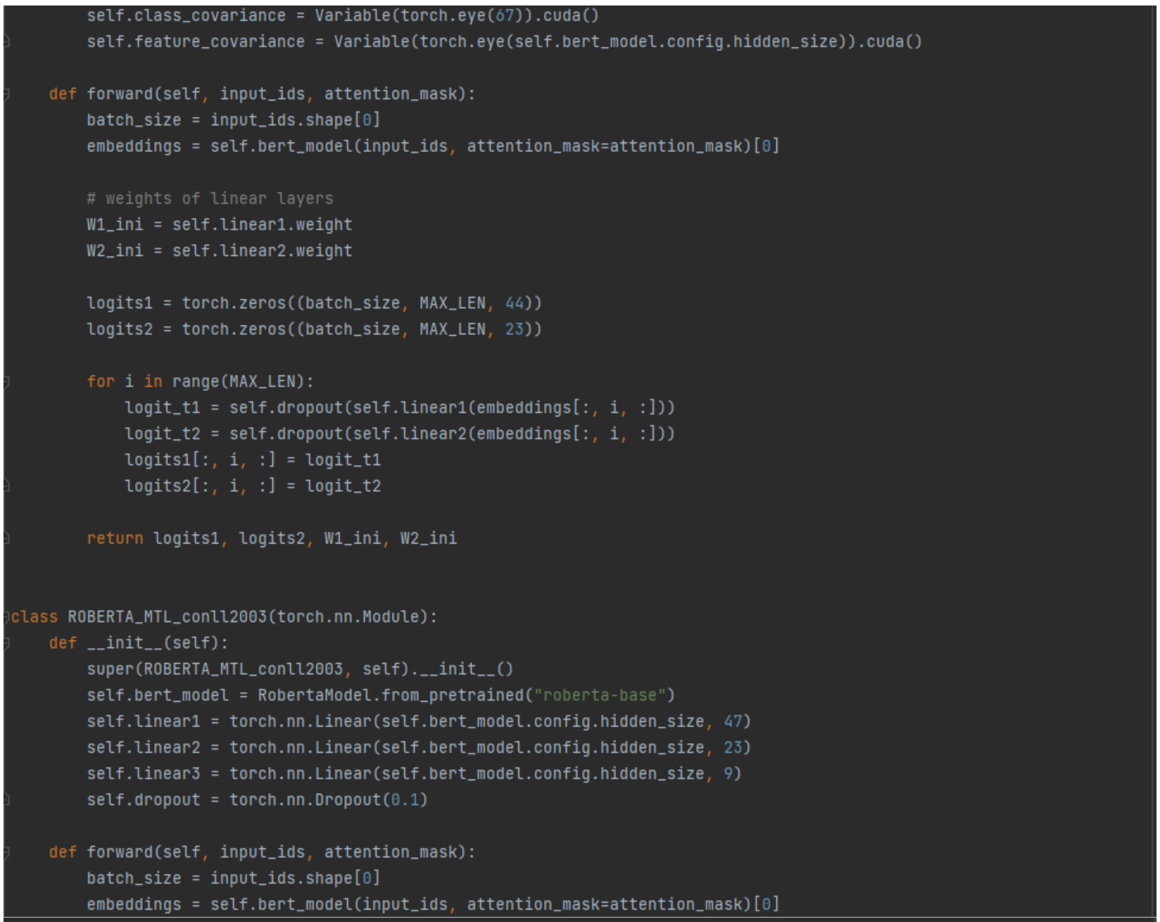
```python
# the code for testing --> make sure the model is tested under the right learning paradigm with the right dataset
# for instance, test_conll2003_MRN_or_MTL() means that the MRN or MTL model will be tested with CoNLL-03 dataset
# Note that test_STL() can be used for both of the datasets

import testers as testers
testers.test_conll2003_MRN_or_MTL(model, test_data_loader, datasets)
# testers.test_conll2000_MRN_or_MTL(model, test_data_loader, datasets)
# testers.test_STL(model, test_data_loader, datasets, task="ner")
# testers.test_both_pos_MRN_or_MTL(model, test_data_loader03, datasets03, test_data_loader00, datasets00)
```

```python
# the code for visualization for the two POS tasks
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
def calculate_correlation(omega):
  print("Covariance Matrix:\n",omega)
  diag = torch.sqrt(torch.diag(torch.diag(omega)))
  gaid = torch.linalg.inv(diag)
  corr = gaid @ omega @ gaid
  print("Correlation Matrix:\n", corr)
  return corr
eps = 1e-5
w = trainers.weights_cat_both_pos(model.linear1.weight, model.linear2.weight)
temp_task_covariance = trainers.covariance_mat_update(w.data, model.class_covariance.data, model.feature_covariance.data)
temp_class_covariance = trainers.covariance_mat_update(w.data.permute(1, 0, 2), model.task_covariance.data, model.feature_covariance.data)
temp_feature_covariance = trainers.covariance_mat_update(w.data.permute(2, 0, 1), model.task_covariance.data, model.class_covariance.data)

print("base_identity_matrix:")
fig = plt.figure(figsize=(10,10))
plt.matshow(torch.eye(2), cmap=plt.get_cmap("seismic"), vmin=-1, vmax=1, fignum=fig.number)
print("task:")
task_fig = plt.figure(figsize=(10,10))
plt.matshow(calculate_correlation(temp_task_covariance.cpu().detach()-eps*torch.eye(2)), cmap=plt.get_cmap("seismic"), vmin=-3, vmax=1, fignum=task_fig.number)
print("class:")
class_fig = plt.figure(figsize=(10,10))
plt.matshow(calculate_correlation(temp_class_covariance.cpu().detach()-eps*torch.eye(47)), cmap=plt.get_cmap("seismic"), vmin=-1, vmax=1, fignum=class_fig.number)
print("feature:")
feature_fig = plt.figure(figsize=(10,10))
plt.matshow(calculate_correlation(temp_feature_covariance.cpu().detach()-eps*torch.eye(768)), cmap=plt.get_cmap("seismic"), vmin=-1, vmax=1, fignum=feature_fig.number)
```

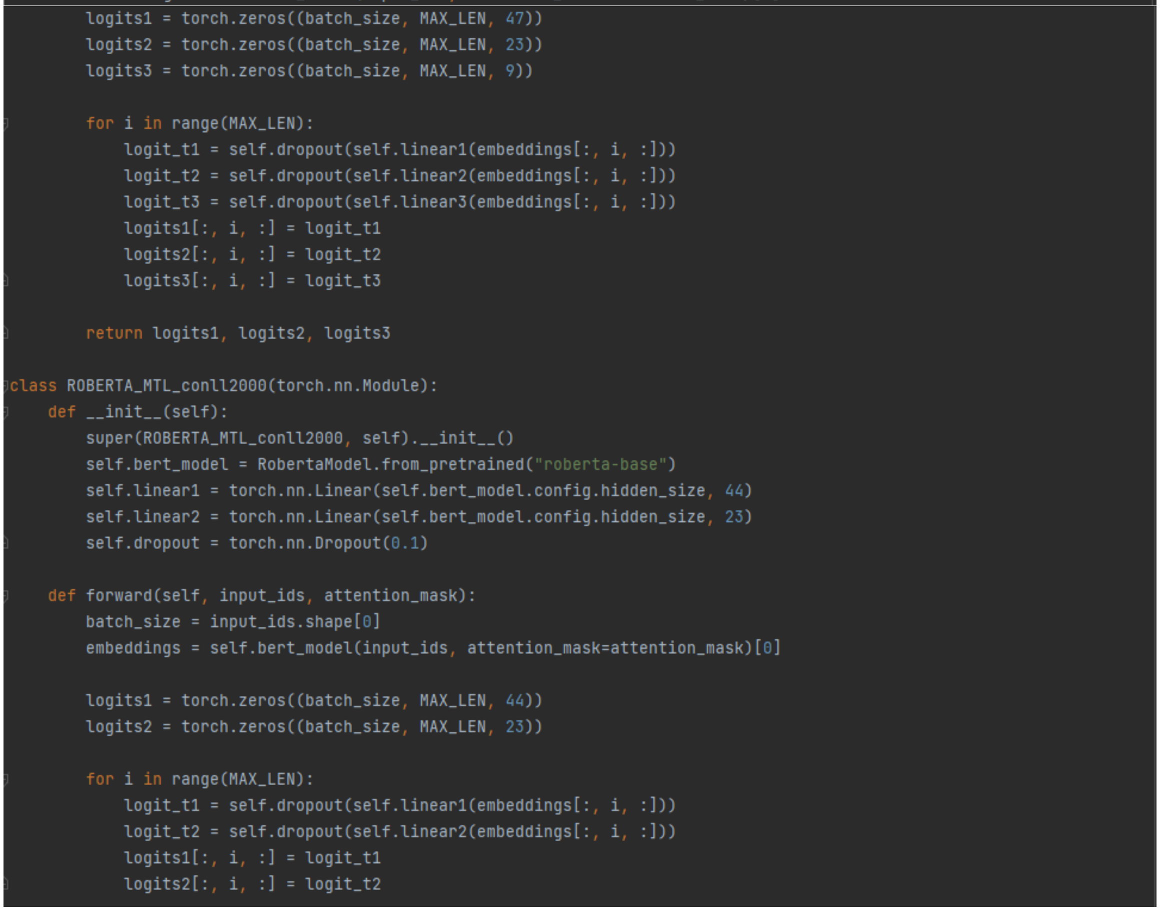
A.2: kcl_individual_project_bert.ipynb



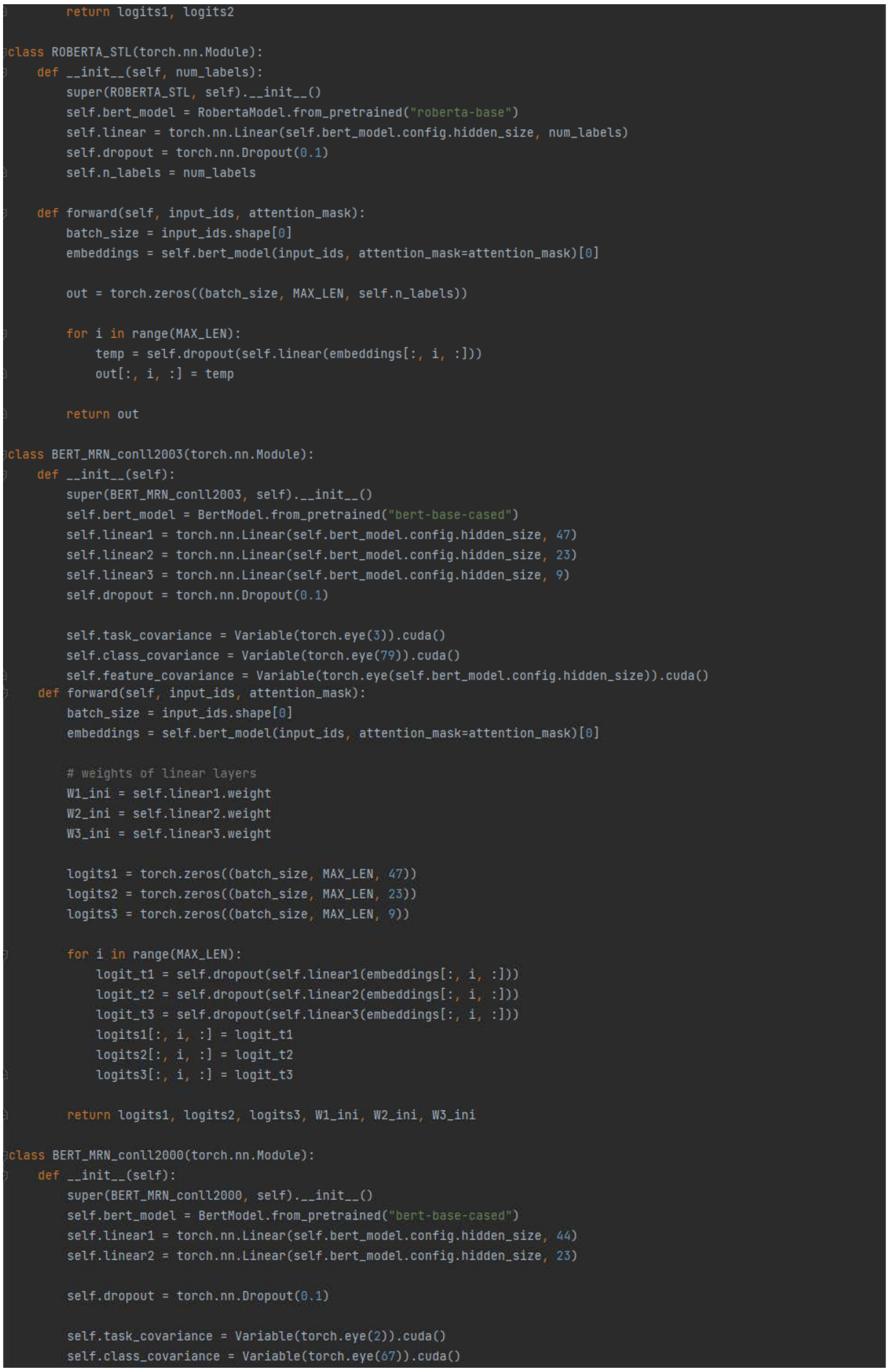
```python
import torch
from transformers import BertModel
from transformers import RobertaModel
from torch.autograd import Variable

MAX_LEN = 256
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

def load_model(checkpoint_path, model, optimizer):
    current_device='cpu'
    if torch.cuda.is_available():
      current_device=None
    checkpoint = torch.load(checkpoint_path, map_location=current_device)
    model.load_state_dict(checkpoint['state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer'])
    epoch = checkpoint['epoch']
    avg_valid_loss = checkpoint['avg_valid_loss']
    return model, optimizer, epoch, avg_valid_loss

def save_model(checkpoint, checkpoint_path):
    torch.save(checkpoint, checkpoint_path)

class ROBERTA_MRN_conll2003(torch.nn.Module):
    def __init__(self):
        super(ROBERTA_MRN_conll2003, self).__init__()
        self.bert_model = RobertaModel.from_pretrained("roberta-base")
        self.linear1 = torch.nn.Linear(self.bert_model.config.hidden_size, 47)
        self.linear2 = torch.nn.Linear(self.bert_model.config.hidden_size, 23)
        self.linear3 = torch.nn.Linear(self.bert_model.config.hidden_size, 9)
        self.dropout = torch.nn.Dropout(0.1)

        self.task_covariance = Variable(torch.eye(3)).cuda()
        self.class_covariance = Variable(torch.eye(79)).cuda()
        self.feature_covariance = Variable(torch.eye(self.bert_model.config.hidden_size)).cuda()

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]

        # weights of linear layers
        W1_ini = self.linear1.weight
        W2_ini = self.linear2.weight
        W3_ini = self.linear3.weight

        logits1 = torch.zeros((batch_size, MAX_LEN, 47))
        logits2 = torch.zeros((batch_size, MAX_LEN, 23))
        logits3 = torch.zeros((batch_size, MAX_LEN, 9))

        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logit_t3 = self.dropout(self.linear3(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2
            logits3[:, i, :] = logit_t3

        return logits1, logits2, logits3, W1_ini, W2_ini, W3_ini

class ROBERTA_MRN_conll2000(torch.nn.Module):
    def __init__(self):
        super(ROBERTA_MRN_conll2000, self).__init__()
        self.bert_model = RobertaModel.from_pretrained("roberta-base")
        self.linear1 = torch.nn.Linear(self.bert_model.config.hidden_size, 44)
        self.linear2 = torch.nn.Linear(self.bert_model.config.hidden_size, 23)

        self.dropout = torch.nn.Dropout(0.1)

        self.task_covariance = Variable(torch.eye(2)).cuda()
        self.class_covariance = Variable(torch.eye(67)).cuda()
```



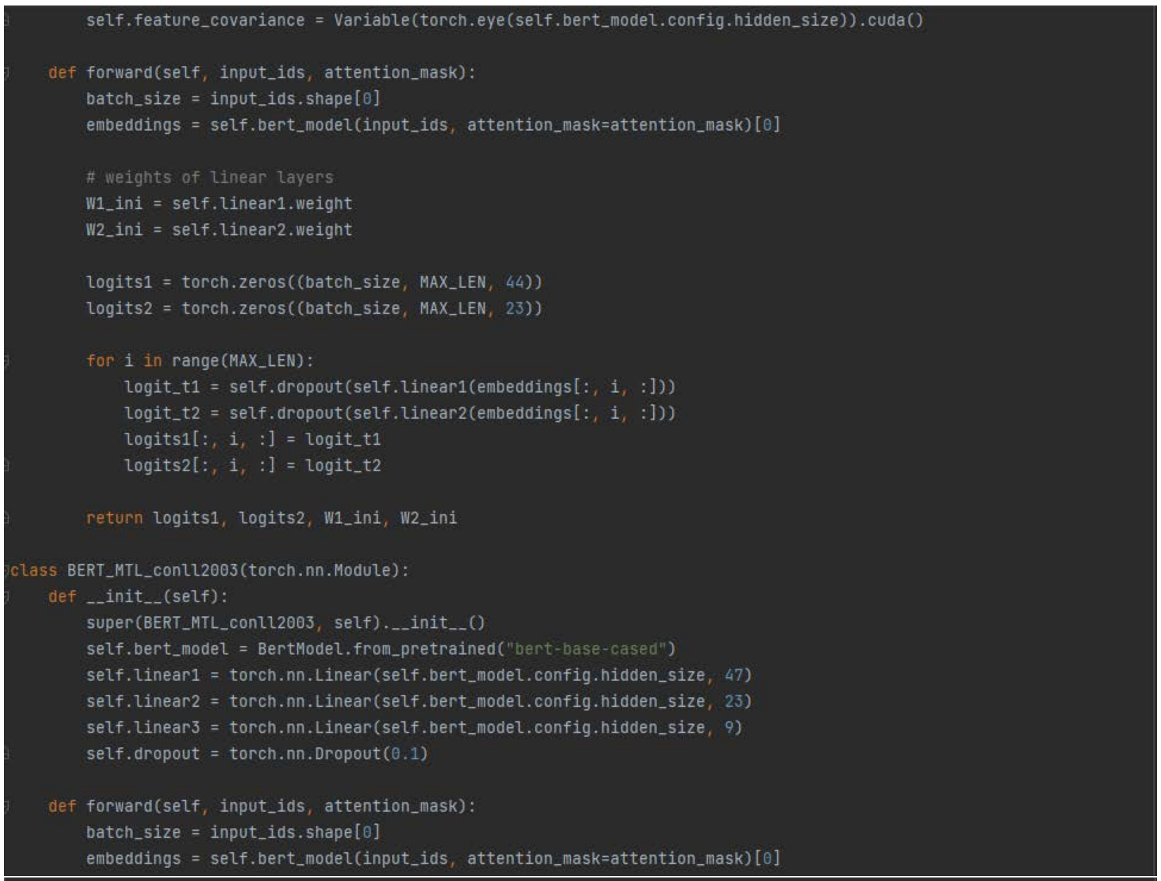
```python
        self.class_covariance = Variable(torch.eye(67)).cuda()
        self.feature_covariance = Variable(torch.eye(self.bert_model.config.hidden_size)).cuda()

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]

        # weights of linear layers
        W1_ini = self.linear1.weight
        W2_ini = self.linear2.weight

        logits1 = torch.zeros((batch_size, MAX_LEN, 44))
        logits2 = torch.zeros((batch_size, MAX_LEN, 23))

        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2

        return logits1, logits2, W1_ini, W2_ini


class ROBERTA_MTL_conll2003(torch.nn.Module):
    def __init__(self):
        super(ROBERTA_MTL_conll2003, self).__init__()
        self.bert_model = RobertaModel.from_pretrained("roberta-base")
        self.linear1 = torch.nn.Linear(self.bert_model.config.hidden_size, 47)
        self.linear2 = torch.nn.Linear(self.bert_model.config.hidden_size, 23)
        self.linear3 = torch.nn.Linear(self.bert_model.config.hidden_size, 9)
        self.dropout = torch.nn.Dropout(0.1)

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]
```



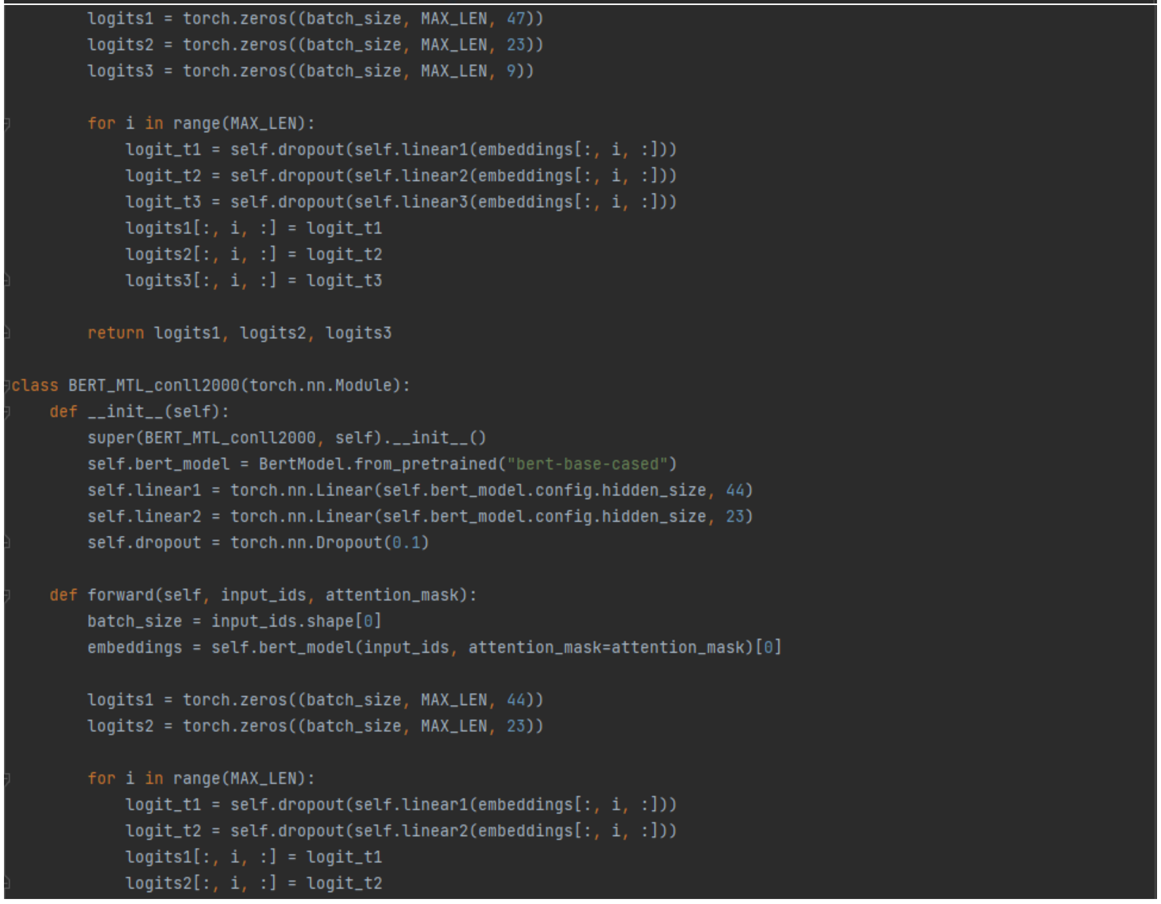
```python
        logits1 = torch.zeros((batch_size, MAX_LEN, 47))
        logits2 = torch.zeros((batch_size, MAX_LEN, 23))
        logits3 = torch.zeros((batch_size, MAX_LEN, 9))

        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logit_t3 = self.dropout(self.linear3(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2
            logits3[:, i, :] = logit_t3

        return logits1, logits2, logits3

class ROBERTA_MTL_conll2000(torch.nn.Module):
    def __init__(self):
        super(ROBERTA_MTL_conll2000, self).__init__()
        self.bert_model = RobertaModel.from_pretrained("roberta-base")
        self.linear1 = torch.nn.Linear(self.bert_model.config.hidden_size, 44)
        self.linear2 = torch.nn.Linear(self.bert_model.config.hidden_size, 23)
        self.dropout = torch.nn.Dropout(0.1)

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]

        logits1 = torch.zeros((batch_size, MAX_LEN, 44))
        logits2 = torch.zeros((batch_size, MAX_LEN, 23))

        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2
```



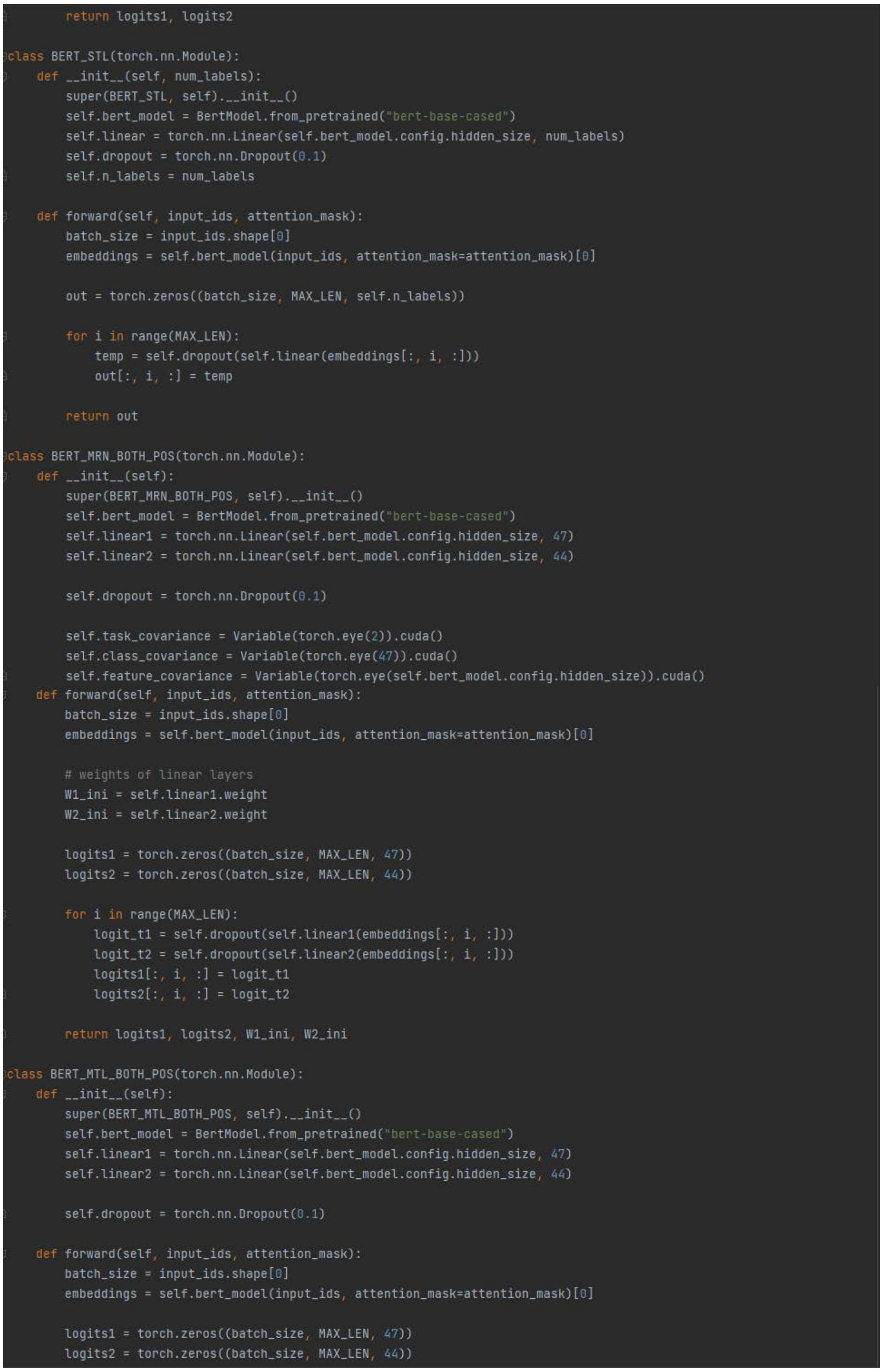
```python
        return logits1, logits2

class ROBERTA_STL(torch.nn.Module):
    def __init__(self, num_labels):
        super(ROBERTA_STL, self).__init__()
        self.bert_model = RobertaModel.from_pretrained("roberta-base")
        self.linear = torch.nn.Linear(self.bert_model.config.hidden_size, num_labels)
        self.dropout = torch.nn.Dropout(0.1)
        self.n_labels = num_labels

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]

        out = torch.zeros((batch_size, MAX_LEN, self.n_labels))

        for i in range(MAX_LEN):
            temp = self.dropout(self.linear(embeddings[:, i, :]))
            out[:, i, :] = temp

        return out

class BERT_MRN_conll2003(torch.nn.Module):
    def __init__(self):
        super(BERT_MRN_conll2003, self).__init__()
        self.bert_model = BertModel.from_pretrained("bert-base-cased")
        self.linear1 = torch.nn.Linear(self.bert_model.config.hidden_size, 47)
        self.linear2 = torch.nn.Linear(self.bert_model.config.hidden_size, 23)
        self.linear3 = torch.nn.Linear(self.bert_model.config.hidden_size, 9)
        self.dropout = torch.nn.Dropout(0.1)

        self.task_covariance = Variable(torch.eye(3)).cuda()
        self.class_covariance = Variable(torch.eye(79)).cuda()
        self.feature_covariance = Variable(torch.eye(self.bert_model.config.hidden_size)).cuda()
    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]

        # weights of linear layers
        W1_ini = self.linear1.weight
        W2_ini = self.linear2.weight
        W3_ini = self.linear3.weight

        logits1 = torch.zeros((batch_size, MAX_LEN, 47))
        logits2 = torch.zeros((batch_size, MAX_LEN, 23))
        logits3 = torch.zeros((batch_size, MAX_LEN, 9))

        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logit_t3 = self.dropout(self.linear3(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2
            logits3[:, i, :] = logit_t3

        return logits1, logits2, logits3, W1_ini, W2_ini, W3_ini

class BERT_MRN_conll2000(torch.nn.Module):
    def __init__(self):
        super(BERT_MRN_conll2000, self).__init__()
        self.bert_model = BertModel.from_pretrained("bert-base-cased")
        self.linear1 = torch.nn.Linear(self.bert_model.config.hidden_size, 44)
        self.linear2 = torch.nn.Linear(self.bert_model.config.hidden_size, 23)

        self.dropout = torch.nn.Dropout(0.1)

        self.task_covariance = Variable(torch.eye(2)).cuda()
        self.class_covariance = Variable(torch.eye(67)).cuda()
```



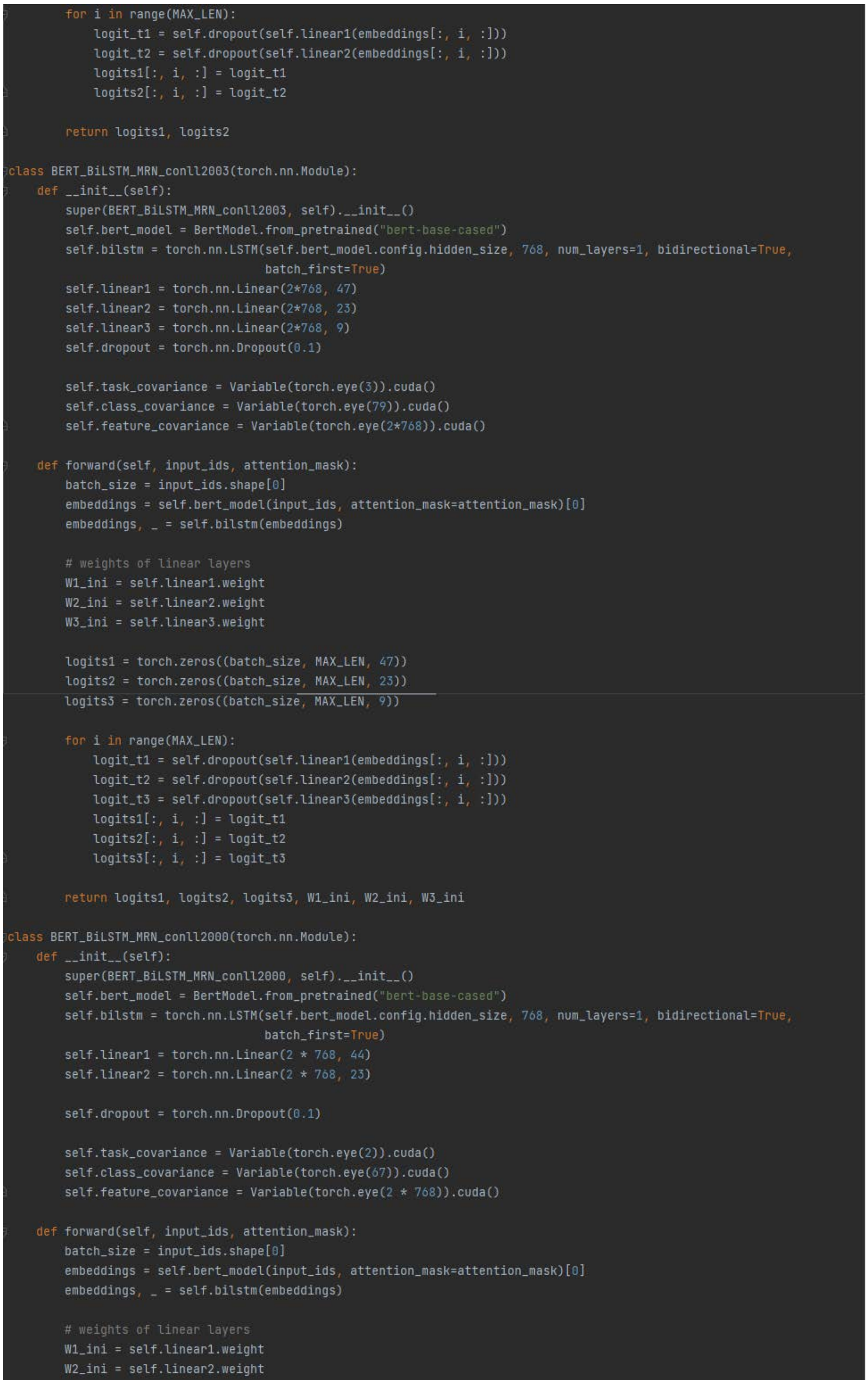
```python
        self.feature_covariance = Variable(torch.eye(self.bert_model.config.hidden_size)).cuda()

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]

        # weights of linear layers
        W1_ini = self.linear1.weight
        W2_ini = self.linear2.weight

        logits1 = torch.zeros((batch_size, MAX_LEN, 44))
        logits2 = torch.zeros((batch_size, MAX_LEN, 23))

        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2

        return logits1, logits2, W1_ini, W2_ini

class BERT_MTL_conll2003(torch.nn.Module):
    def __init__(self):
        super(BERT_MTL_conll2003, self).__init__()
        self.bert_model = BertModel.from_pretrained("bert-base-cased")
        self.linear1 = torch.nn.Linear(self.bert_model.config.hidden_size, 47)
        self.linear2 = torch.nn.Linear(self.bert_model.config.hidden_size, 23)
        self.linear3 = torch.nn.Linear(self.bert_model.config.hidden_size, 9)
        self.dropout = torch.nn.Dropout(0.1)

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]
```



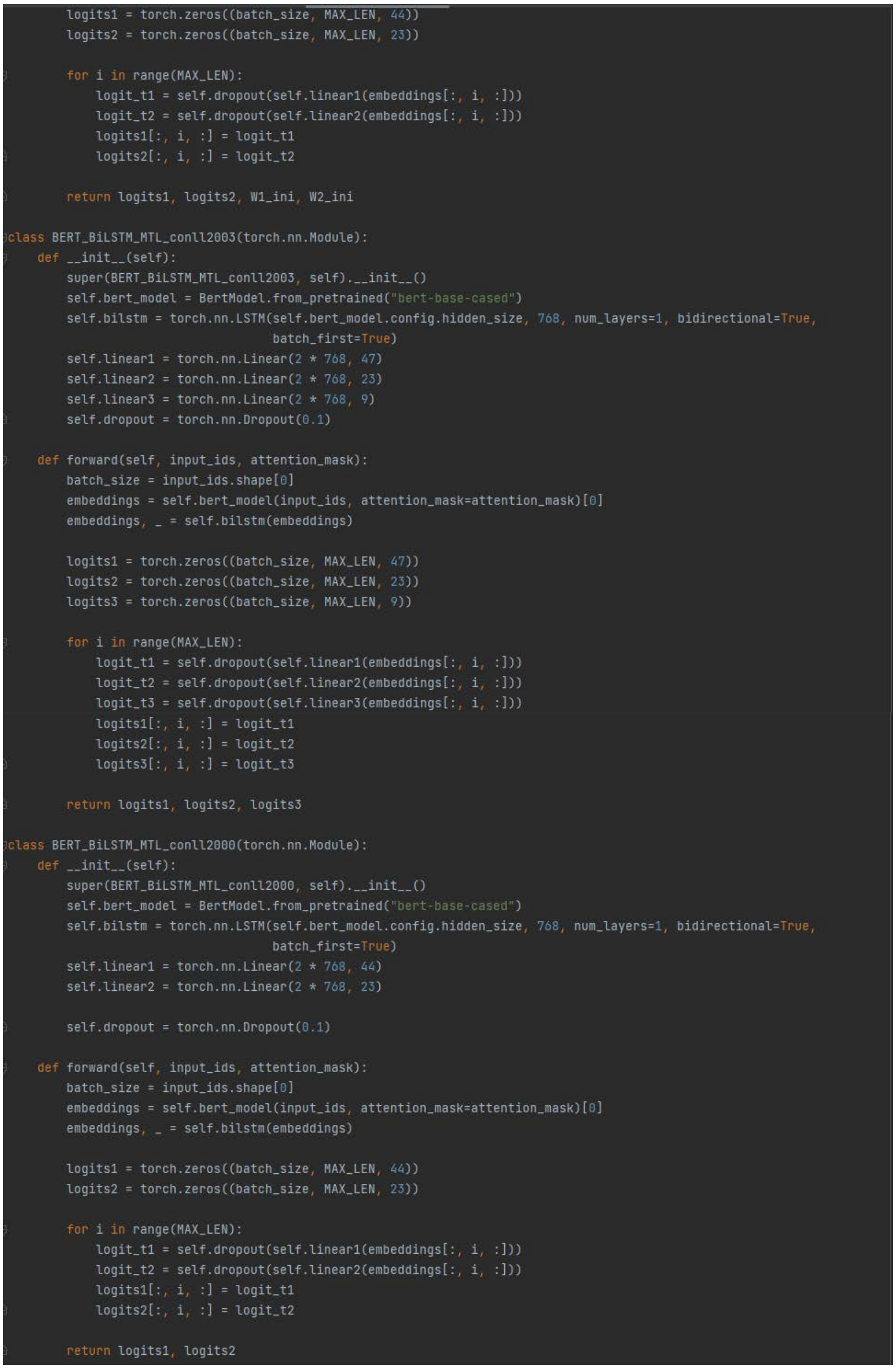
```python
        logits1 = torch.zeros((batch_size, MAX_LEN, 47))
        logits2 = torch.zeros((batch_size, MAX_LEN, 23))
        logits3 = torch.zeros((batch_size, MAX_LEN, 9))

        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logit_t3 = self.dropout(self.linear3(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2
            logits3[:, i, :] = logit_t3

        return logits1, logits2, logits3

class BERT_MTL_conll2000(torch.nn.Module):
    def __init__(self):
        super(BERT_MTL_conll2000, self).__init__()
        self.bert_model = BertModel.from_pretrained("bert-base-cased")
        self.linear1 = torch.nn.Linear(self.bert_model.config.hidden_size, 44)
        self.linear2 = torch.nn.Linear(self.bert_model.config.hidden_size, 23)
        self.dropout = torch.nn.Dropout(0.1)

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]

        logits1 = torch.zeros((batch_size, MAX_LEN, 44))
        logits2 = torch.zeros((batch_size, MAX_LEN, 23))

        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2
```



```
        return logits1, logits2

class BERT_STL(torch.nn.Module):
    def __init__(self, num_labels):
        super(BERT_STL, self).__init__()
        self.bert_model = BertModel.from_pretrained("bert-base-cased")
        self.linear = torch.nn.Linear(self.bert_model.config.hidden_size, num_labels)
        self.dropout = torch.nn.Dropout(0.1)
        self.n_labels = num_labels

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]

        out = torch.zeros((batch_size, MAX_LEN, self.n_labels))

        for i in range(MAX_LEN):
            temp = self.dropout(self.linear(embeddings[:, i, :]))
            out[:, i, :] = temp

        return out

class BERT_MRN_BOTH_POS(torch.nn.Module):
    def __init__(self):
        super(BERT_MRN_BOTH_POS, self).__init__()
        self.bert_model = BertModel.from_pretrained("bert-base-cased")
        self.linear1 = torch.nn.Linear(self.bert_model.config.hidden_size, 47)
        self.linear2 = torch.nn.Linear(self.bert_model.config.hidden_size, 44)

        self.dropout = torch.nn.Dropout(0.1)

        self.task_covariance = Variable(torch.eye(2)).cuda()
        self.class_covariance = Variable(torch.eye(47)).cuda()
        self.feature_covariance = Variable(torch.eye(self.bert_model.config.hidden_size)).cuda()
    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]

        # weights of linear layers
        W1_ini = self.linear1.weight
        W2_ini = self.linear2.weight

        logits1 = torch.zeros((batch_size, MAX_LEN, 47))
        logits2 = torch.zeros((batch_size, MAX_LEN, 44))

        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2

        return logits1, logits2, W1_ini, W2_ini

class BERT_MTL_BOTH_POS(torch.nn.Module):
    def __init__(self):
        super(BERT_MTL_BOTH_POS, self).__init__()
        self.bert_model = BertModel.from_pretrained("bert-base-cased")
        self.linear1 = torch.nn.Linear(self.bert_model.config.hidden_size, 47)
        self.linear2 = torch.nn.Linear(self.bert_model.config.hidden_size, 44)

        self.dropout = torch.nn.Dropout(0.1)

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]

        logits1 = torch.zeros((batch_size, MAX_LEN, 47))
        logits2 = torch.zeros((batch_size, MAX_LEN, 44))
```

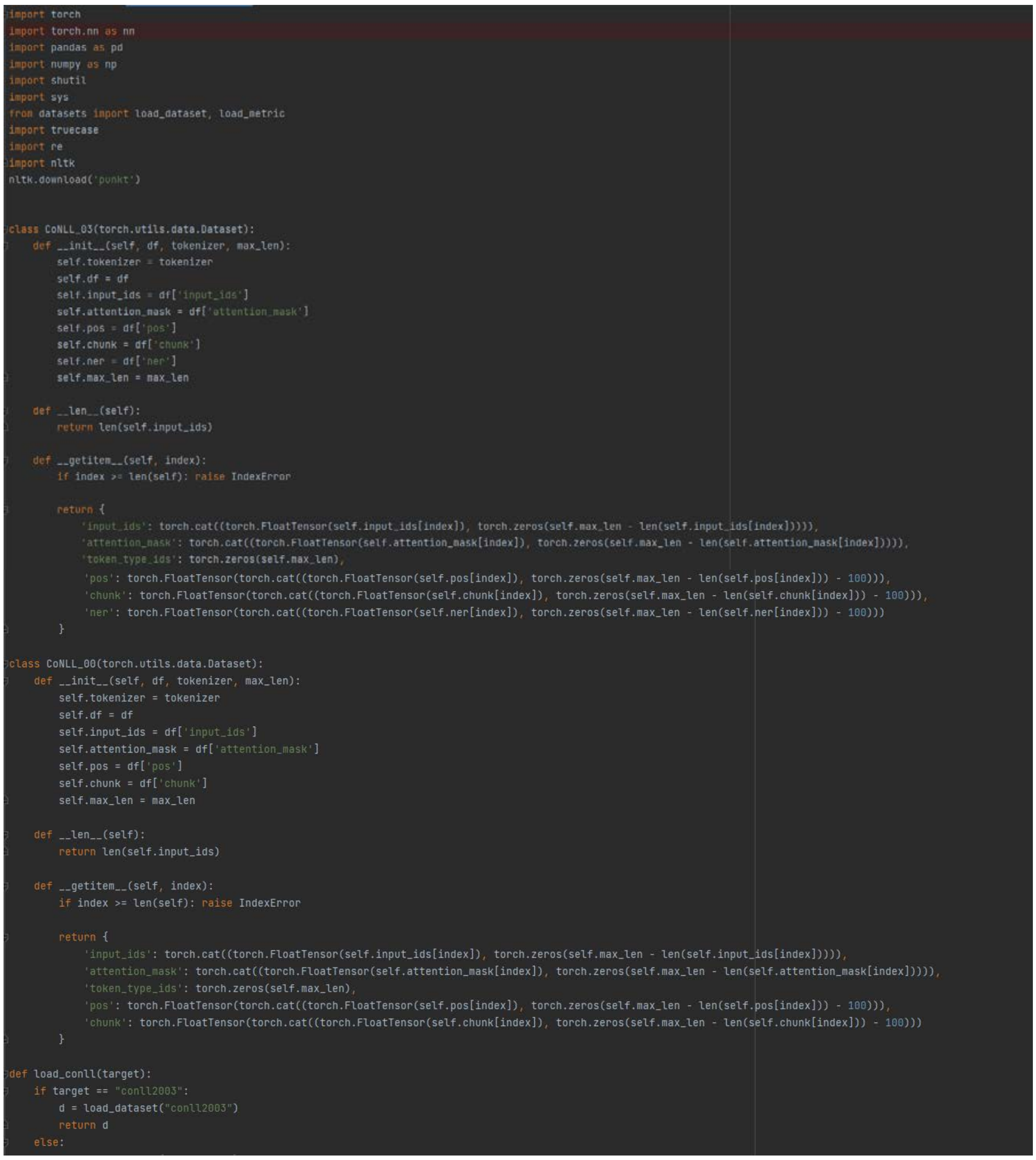


```python
        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2

        return logits1, logits2

class BERT_BiLSTM_MRN_conll2003(torch.nn.Module):
    def __init__(self):
        super(BERT_BiLSTM_MRN_conll2003, self).__init__()
        self.bert_model = BertModel.from_pretrained("bert-base-cased")
        self.bilstm = torch.nn.LSTM(self.bert_model.config.hidden_size, 768, num_layers=1, bidirectional=True,
                                    batch_first=True)
        self.linear1 = torch.nn.Linear(2*768, 47)
        self.linear2 = torch.nn.Linear(2*768, 23)
        self.linear3 = torch.nn.Linear(2*768, 9)
        self.dropout = torch.nn.Dropout(0.1)

        self.task_covariance = Variable(torch.eye(3)).cuda()
        self.class_covariance = Variable(torch.eye(79)).cuda()
        self.feature_covariance = Variable(torch.eye(2*768)).cuda()

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]
        embeddings, _ = self.bilstm(embeddings)

        # weights of linear layers
        W1_ini = self.linear1.weight
        W2_ini = self.linear2.weight
        W3_ini = self.linear3.weight

        logits1 = torch.zeros((batch_size, MAX_LEN, 47))
        logits2 = torch.zeros((batch_size, MAX_LEN, 23))
        logits3 = torch.zeros((batch_size, MAX_LEN, 9))

        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logit_t3 = self.dropout(self.linear3(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2
            logits3[:, i, :] = logit_t3

        return logits1, logits2, logits3, W1_ini, W2_ini, W3_ini

class BERT_BiLSTM_MRN_conll2000(torch.nn.Module):
    def __init__(self):
        super(BERT_BiLSTM_MRN_conll2000, self).__init__()
        self.bert_model = BertModel.from_pretrained("bert-base-cased")
        self.bilstm = torch.nn.LSTM(self.bert_model.config.hidden_size, 768, num_layers=1, bidirectional=True,
                                    batch_first=True)
        self.linear1 = torch.nn.Linear(2 * 768, 44)
        self.linear2 = torch.nn.Linear(2 * 768, 23)

        self.dropout = torch.nn.Dropout(0.1)

        self.task_covariance = Variable(torch.eye(2)).cuda()
        self.class_covariance = Variable(torch.eye(67)).cuda()
        self.feature_covariance = Variable(torch.eye(2 * 768)).cuda()

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]
        embeddings, _ = self.bilstm(embeddings)

        # weights of linear layers
        W1_ini = self.linear1.weight
        W2_ini = self.linear2.weight
```

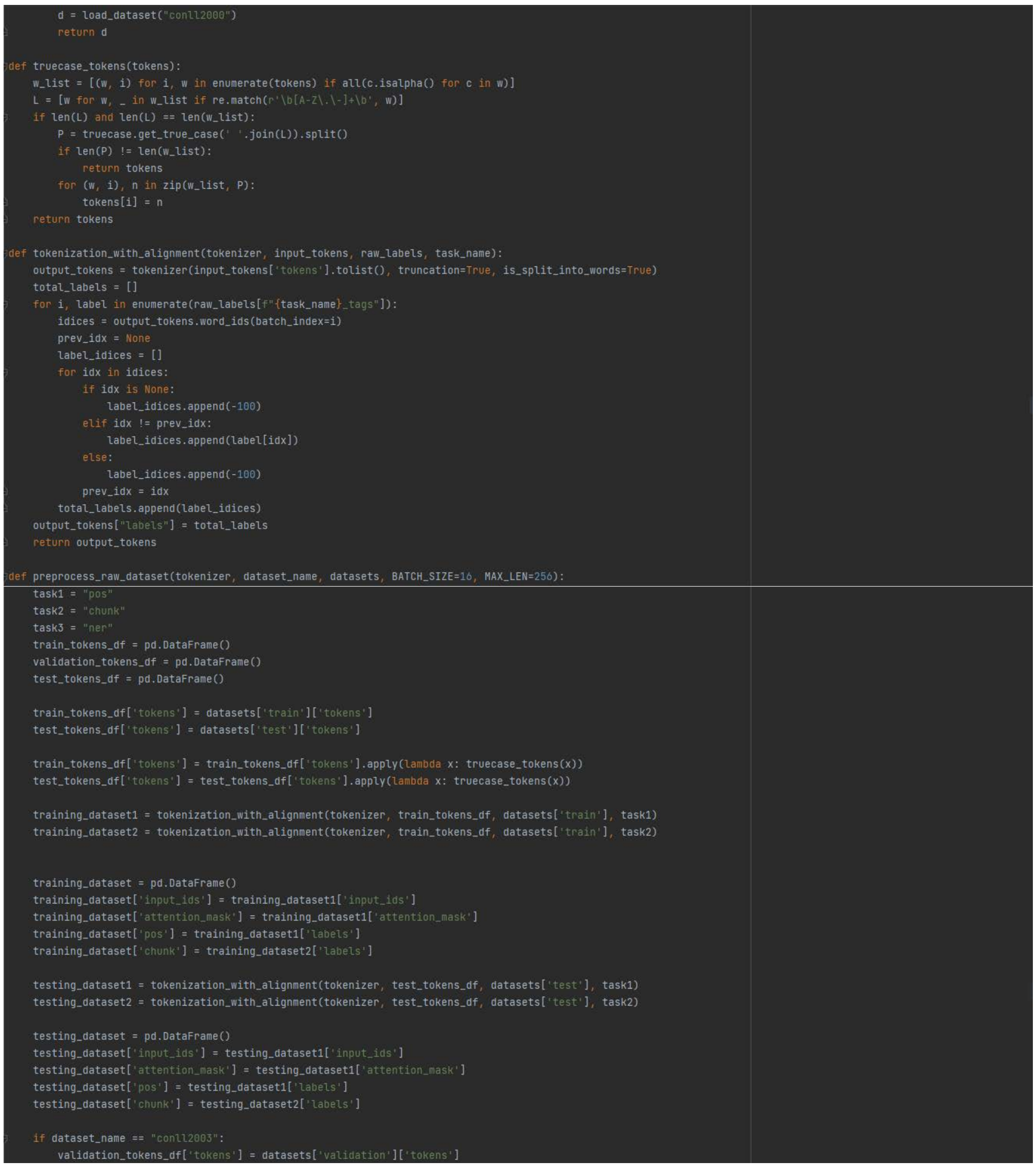


```python
        logits1 = torch.zeros((batch_size, MAX_LEN, 44))
        logits2 = torch.zeros((batch_size, MAX_LEN, 23))

        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2

        return logits1, logits2, W1_ini, W2_ini

class BERT_BiLSTM_MTL_conll2003(torch.nn.Module):
    def __init__(self):
        super(BERT_BiLSTM_MTL_conll2003, self).__init__()
        self.bert_model = BertModel.from_pretrained("bert-base-cased")
        self.bilstm = torch.nn.LSTM(self.bert_model.config.hidden_size, 768, num_layers=1, bidirectional=True,
                                    batch_first=True)
        self.linear1 = torch.nn.Linear(2 * 768, 47)
        self.linear2 = torch.nn.Linear(2 * 768, 23)
        self.linear3 = torch.nn.Linear(2 * 768, 9)
        self.dropout = torch.nn.Dropout(0.1)

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]
        embeddings, _ = self.bilstm(embeddings)

        logits1 = torch.zeros((batch_size, MAX_LEN, 47))
        logits2 = torch.zeros((batch_size, MAX_LEN, 23))
        logits3 = torch.zeros((batch_size, MAX_LEN, 9))

        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logit_t3 = self.dropout(self.linear3(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2
            logits3[:, i, :] = logit_t3

        return logits1, logits2, logits3

class BERT_BiLSTM_MTL_conll2000(torch.nn.Module):
    def __init__(self):
        super(BERT_BiLSTM_MTL_conll2000, self).__init__()
        self.bert_model = BertModel.from_pretrained("bert-base-cased")
        self.bilstm = torch.nn.LSTM(self.bert_model.config.hidden_size, 768, num_layers=1, bidirectional=True,
                                    batch_first=True)
        self.linear1 = torch.nn.Linear(2 * 768, 44)
        self.linear2 = torch.nn.Linear(2 * 768, 23)

        self.dropout = torch.nn.Dropout(0.1)

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.shape[0]
        embeddings = self.bert_model(input_ids, attention_mask=attention_mask)[0]
        embeddings, _ = self.bilstm(embeddings)

        logits1 = torch.zeros((batch_size, MAX_LEN, 44))
        logits2 = torch.zeros((batch_size, MAX_LEN, 23))

        for i in range(MAX_LEN):
            logit_t1 = self.dropout(self.linear1(embeddings[:, i, :]))
            logit_t2 = self.dropout(self.linear2(embeddings[:, i, :]))
            logits1[:, i, :] = logit_t1
            logits2[:, i, :] = logit_t2

        return logits1, logits2
```

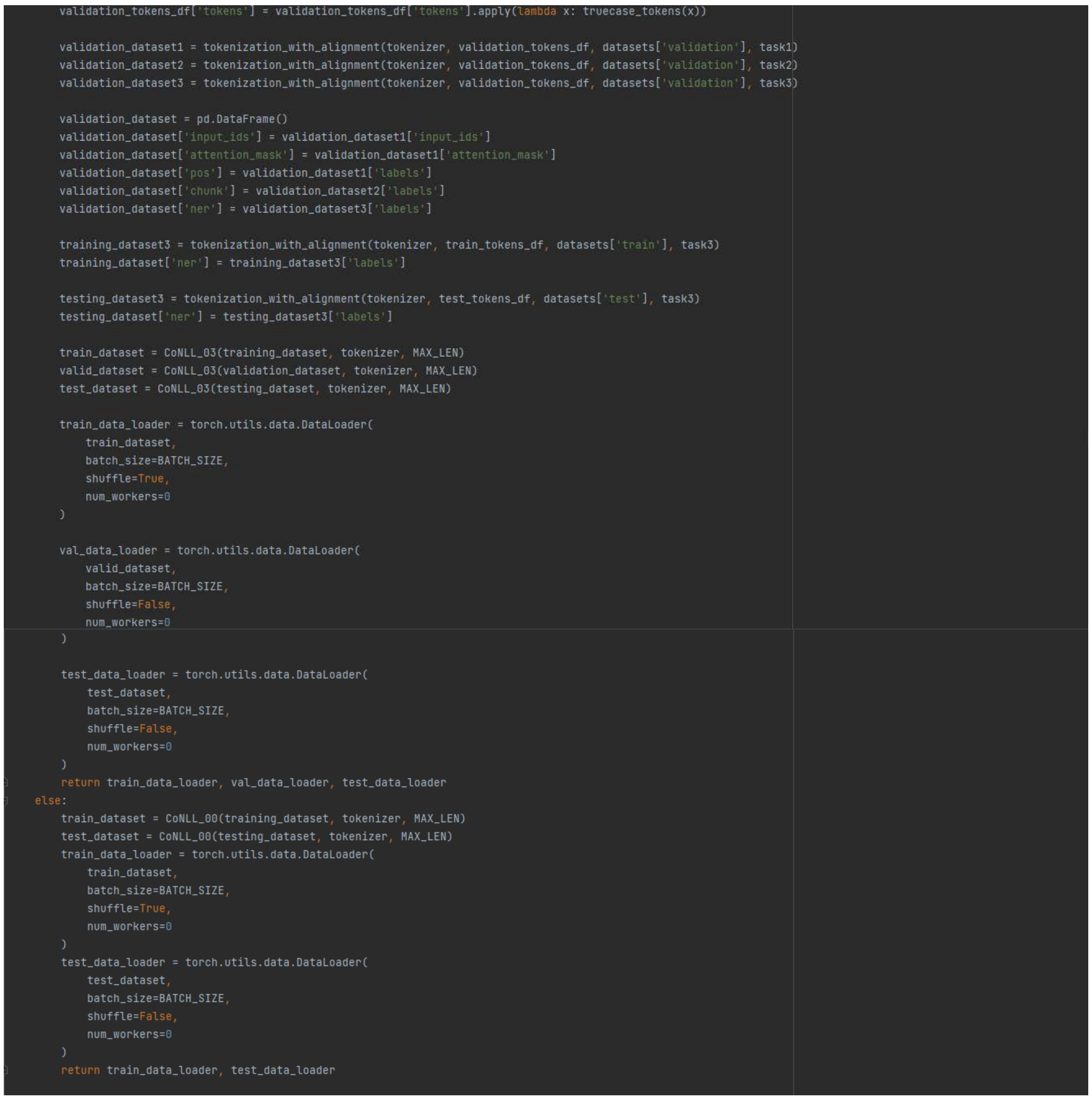
A.3: comp_models.py



```python
import torch
import torch.nn as nn
import pandas as pd
import numpy as np
import shutil
import sys
from datasets import load_dataset, load_metric
import truecase
import re
import nltk
nltk.download('punkt')


class CoNLL_03(torch.utils.data.Dataset):
    def __init__(self, df, tokenizer, max_len):
        self.tokenizer = tokenizer
        self.df = df
        self.input_ids = df['input_ids']
        self.attention_mask = df['attention_mask']
        self.pos = df['pos']
        self.chunk = df['chunk']
        self.ner = df['ner']
        self.max_len = max_len

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, index):
        if index >= len(self): raise IndexError

        return {
            'input_ids': torch.cat((torch.FloatTensor(self.input_ids[index]), torch.zeros(self.max_len - len(self.input_ids[index])))),
            'attention_mask': torch.cat((torch.FloatTensor(self.attention_mask[index]), torch.zeros(self.max_len - len(self.attention_mask[index])))),
            'token_type_ids': torch.zeros(self.max_len),
            'pos': torch.FloatTensor(torch.cat((torch.FloatTensor(self.pos[index]), torch.zeros(self.max_len - len(self.pos[index])) - 100))),
            'chunk': torch.FloatTensor(torch.cat((torch.FloatTensor(self.chunk[index]), torch.zeros(self.max_len - len(self.chunk[index])) - 100))),
            'ner': torch.FloatTensor(torch.cat((torch.FloatTensor(self.ner[index]), torch.zeros(self.max_len - len(self.ner[index])) - 100)))
        }

class CoNLL_00(torch.utils.data.Dataset):
    def __init__(self, df, tokenizer, max_len):
        self.tokenizer = tokenizer
        self.df = df
        self.input_ids = df['input_ids']
        self.attention_mask = df['attention_mask']
        self.pos = df['pos']
        self.chunk = df['chunk']
        self.max_len = max_len

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, index):
        if index >= len(self): raise IndexError

        return {
            'input_ids': torch.cat((torch.FloatTensor(self.input_ids[index]), torch.zeros(self.max_len - len(self.input_ids[index])))),
            'attention_mask': torch.cat((torch.FloatTensor(self.attention_mask[index]), torch.zeros(self.max_len - len(self.attention_mask[index])))),
            'token_type_ids': torch.zeros(self.max_len),
            'pos': torch.FloatTensor(torch.cat((torch.FloatTensor(self.pos[index]), torch.zeros(self.max_len - len(self.pos[index])) - 100))),
            'chunk': torch.FloatTensor(torch.cat((torch.FloatTensor(self.chunk[index]), torch.zeros(self.max_len - len(self.chunk[index])) - 100)))
        }

def load_conll(target):
    if target == "conll2003":
        d = load_dataset("conll2003")
        return d
    else:
```

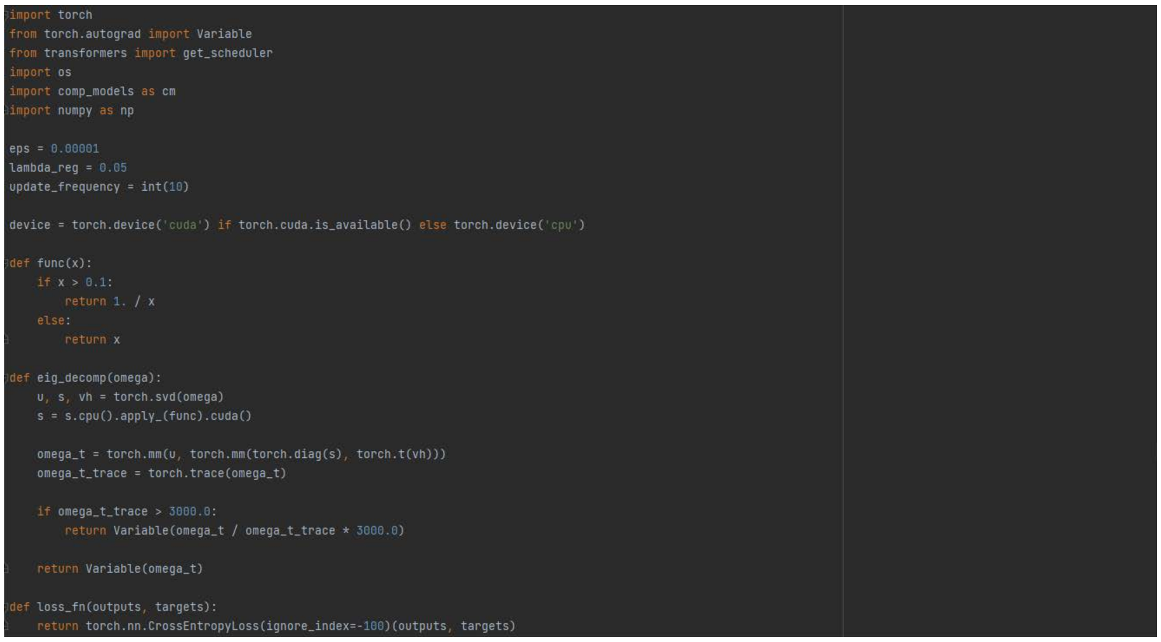


```python
        d = load_dataset("conll2000")
        return d

def truecase_tokens(tokens):
    w_list = [(w, i) for i, w in enumerate(tokens) if all(c.isalpha() for c in w)]
    L = [w for w, _ in w_list if re.match(r'\b[A-Z\.\-]+\b', w)]
    if len(L) and len(L) == len(w_list):
        P = truecase.get_true_case(' '.join(L)).split()
        if len(P) != len(w_list):
            return tokens
        for (w, i), n in zip(w_list, P):
            tokens[i] = n
    return tokens

def tokenization_with_alignment(tokenizer, input_tokens, raw_labels, task_name):
    output_tokens = tokenizer(input_tokens['tokens'].tolist(), truncation=True, is_split_into_words=True)
    total_labels = []
    for i, label in enumerate(raw_labels[f"{task_name}_tags"]):
        idices = output_tokens.word_ids(batch_index=i)
        prev_idx = None
        label_idices = []
        for idx in idices:
            if idx is None:
                label_idices.append(-100)
            elif idx != prev_idx:
                label_idices.append(label[idx])
            else:
                label_idices.append(-100)
            prev_idx = idx
        total_labels.append(label_idices)
    output_tokens["labels"] = total_labels
    return output_tokens

def preprocess_raw_dataset(tokenizer, dataset_name, datasets, BATCH_SIZE=16, MAX_LEN=256):
    task1 = "pos"
    task2 = "chunk"
    task3 = "ner"
    train_tokens_df = pd.DataFrame()
    validation_tokens_df = pd.DataFrame()
    test_tokens_df = pd.DataFrame()

    train_tokens_df['tokens'] = datasets['train']['tokens']
    test_tokens_df['tokens'] = datasets['test']['tokens']

    train_tokens_df['tokens'] = train_tokens_df['tokens'].apply(lambda x: truecase_tokens(x))
    test_tokens_df['tokens'] = test_tokens_df['tokens'].apply(lambda x: truecase_tokens(x))

    training_dataset1 = tokenization_with_alignment(tokenizer, train_tokens_df, datasets['train'], task1)
    training_dataset2 = tokenization_with_alignment(tokenizer, train_tokens_df, datasets['train'], task2)


    training_dataset = pd.DataFrame()
    training_dataset['input_ids'] = training_dataset1['input_ids']
    training_dataset['attention_mask'] = training_dataset1['attention_mask']
    training_dataset['pos'] = training_dataset1['labels']
    training_dataset['chunk'] = training_dataset2['labels']

    testing_dataset1 = tokenization_with_alignment(tokenizer, test_tokens_df, datasets['test'], task1)
    testing_dataset2 = tokenization_with_alignment(tokenizer, test_tokens_df, datasets['test'], task2)

    testing_dataset = pd.DataFrame()
    testing_dataset['input_ids'] = testing_dataset1['input_ids']
    testing_dataset['attention_mask'] = testing_dataset1['attention_mask']
    testing_dataset['pos'] = testing_dataset1['labels']
    testing_dataset['chunk'] = testing_dataset2['labels']

    if dataset_name == "conll2003":
        validation_tokens_df['tokens'] = datasets['validation']['tokens']
```

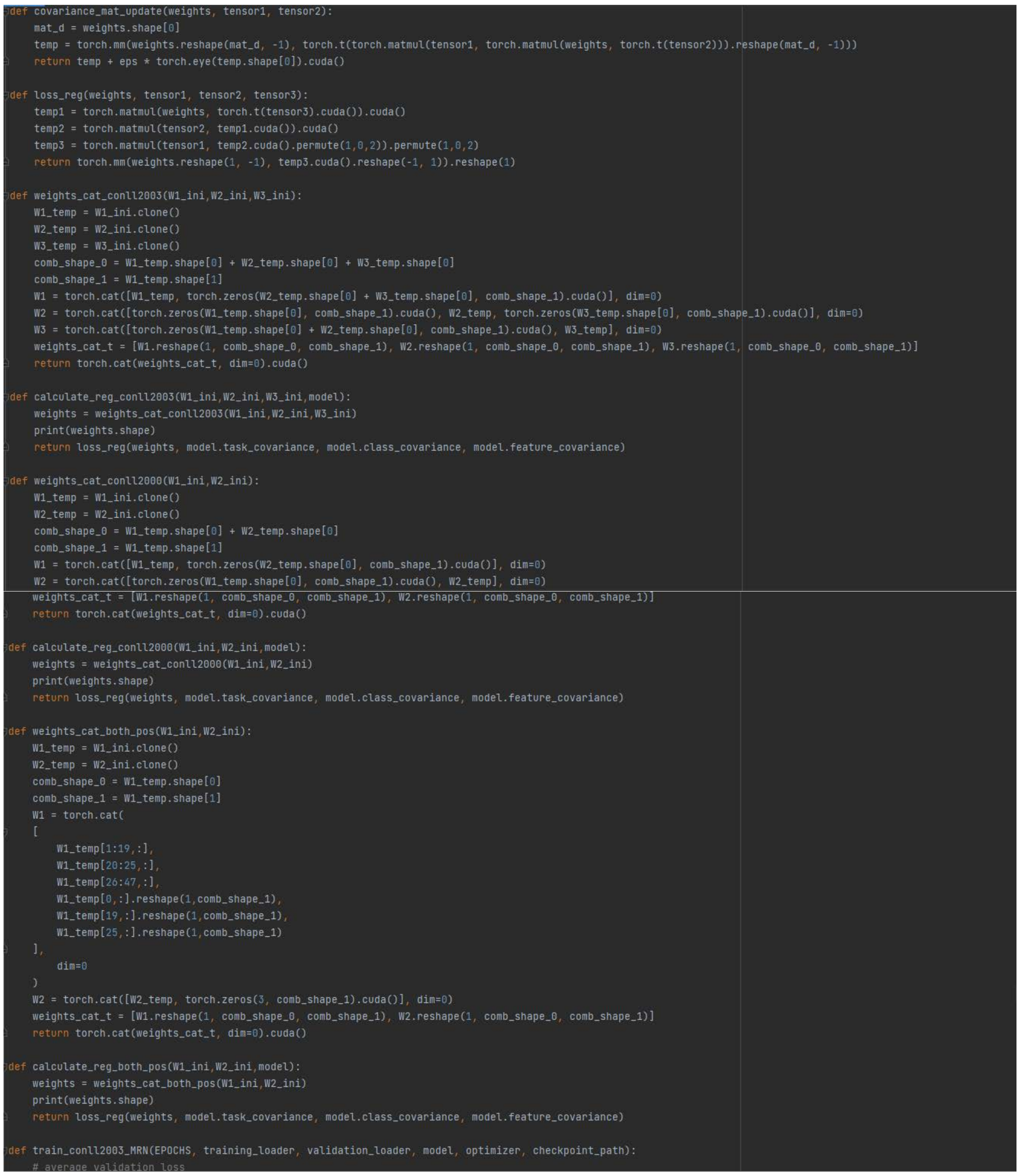


```python
        validation_tokens_df['tokens'] = validation_tokens_df['tokens'].apply(lambda x: truecase_tokens(x))

        validation_dataset1 = tokenization_with_alignment(tokenizer, validation_tokens_df, datasets['validation'], task1)
        validation_dataset2 = tokenization_with_alignment(tokenizer, validation_tokens_df, datasets['validation'], task2)
        validation_dataset3 = tokenization_with_alignment(tokenizer, validation_tokens_df, datasets['validation'], task3)

        validation_dataset = pd.DataFrame()
        validation_dataset['input_ids'] = validation_dataset1['input_ids']
        validation_dataset['attention_mask'] = validation_dataset1['attention_mask']
        validation_dataset['pos'] = validation_dataset1['labels']
        validation_dataset['chunk'] = validation_dataset2['labels']
        validation_dataset['ner'] = validation_dataset3['labels']

        training_dataset3 = tokenization_with_alignment(tokenizer, train_tokens_df, datasets['train'], task3)
        training_dataset['ner'] = training_dataset3['labels']

        testing_dataset3 = tokenization_with_alignment(tokenizer, test_tokens_df, datasets['test'], task3)
        testing_dataset['ner'] = testing_dataset3['labels']

        train_dataset = CoNLL_03(training_dataset, tokenizer, MAX_LEN)
        valid_dataset = CoNLL_03(validation_dataset, tokenizer, MAX_LEN)
        test_dataset = CoNLL_03(testing_dataset, tokenizer, MAX_LEN)

        train_data_loader = torch.utils.data.DataLoader(
            train_dataset,
            batch_size=BATCH_SIZE,
            shuffle=True,
            num_workers=0
        )

        val_data_loader = torch.utils.data.DataLoader(
            valid_dataset,
            batch_size=BATCH_SIZE,
            shuffle=False,
            num_workers=0
        )

        test_data_loader = torch.utils.data.DataLoader(
            test_dataset,
            batch_size=BATCH_SIZE,
            shuffle=False,
            num_workers=0
        )
        return train_data_loader, val_data_loader, test_data_loader
    else:
        train_dataset = CoNLL_00(training_dataset, tokenizer, MAX_LEN)
        test_dataset = CoNLL_00(testing_dataset, tokenizer, MAX_LEN)
        train_data_loader = torch.utils.data.DataLoader(
            train_dataset,
            batch_size=BATCH_SIZE,
            shuffle=True,
            num_workers=0
        )
        test_data_loader = torch.utils.data.DataLoader(
            test_dataset,
            batch_size=BATCH_SIZE,
            shuffle=False,
            num_workers=0
        )
        return train_data_loader, test_data_loader
```

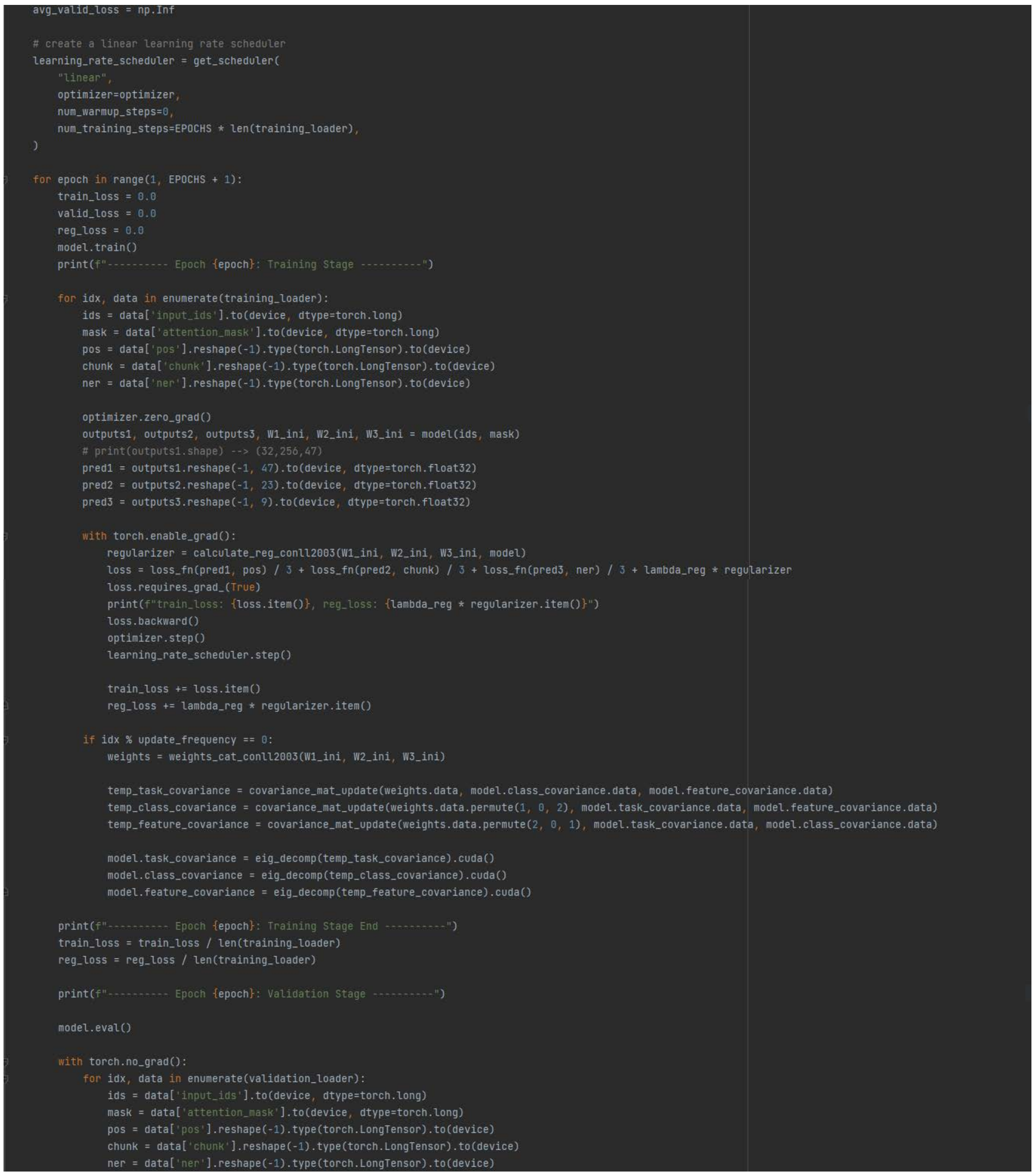
A.4: data_preprocess.py



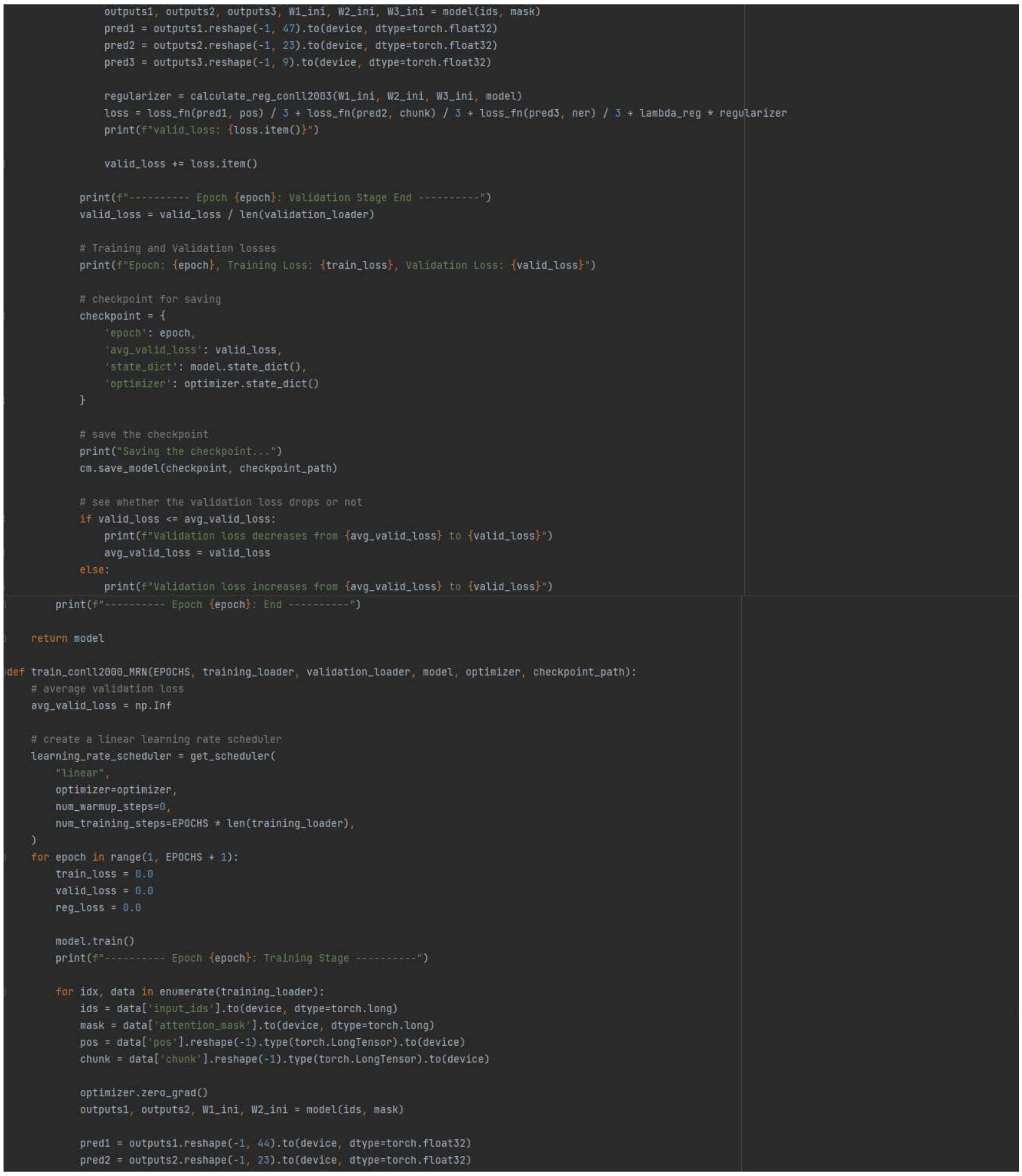
```python
import torch
from torch.autograd import Variable
from transformers import get_scheduler
import os
import comp_models as cm
import numpy as np

eps = 0.00001
lambda_reg = 0.05
update_frequency = int(10)

device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

def func(x):
    if x > 0.1:
        return 1. / x
    else:
        return x

def eig_decomp(omega):
    u, s, vh = torch.svd(omega)
    s = s.cpu().apply_(func).cuda()

    omega_t = torch.mm(u, torch.mm(torch.diag(s), torch.t(vh)))
    omega_t_trace = torch.trace(omega_t)

    if omega_t_trace > 3000.0:
        return Variable(omega_t / omega_t_trace * 3000.0)

    return Variable(omega_t)

def loss_fn(outputs, targets):
    return torch.nn.CrossEntropyLoss(ignore_index=-100)(outputs, targets)
```



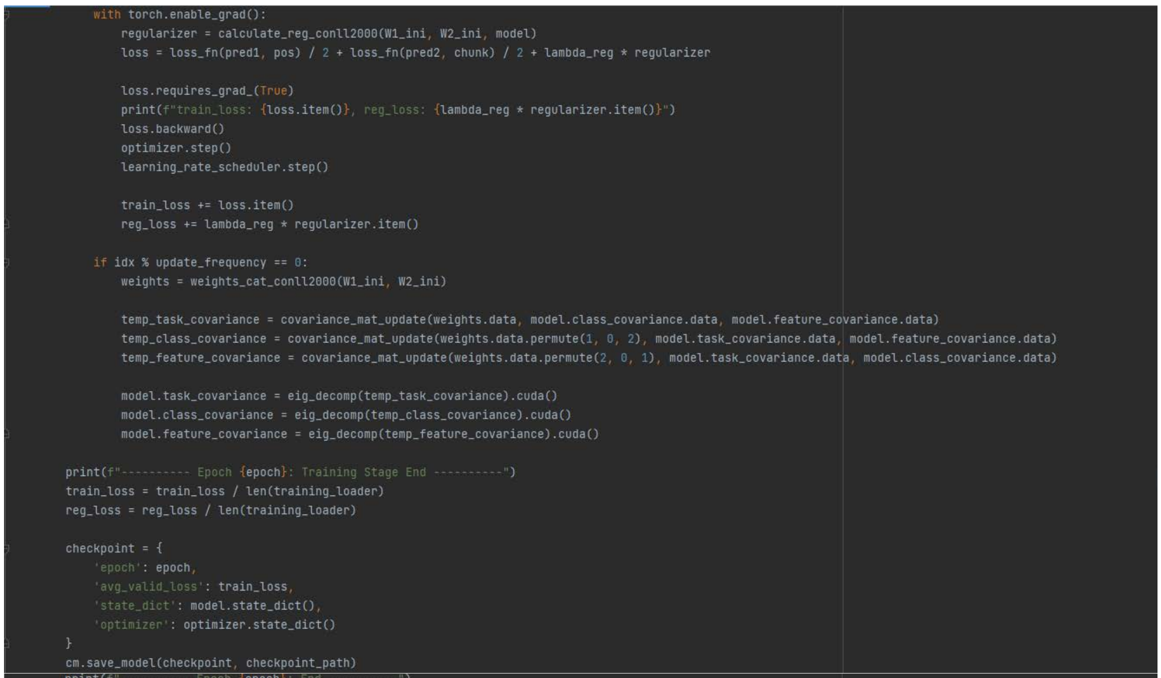
```python
def covariance_mat_update(weights, tensor1, tensor2):
    mat_d = weights.shape[0]
    temp = torch.mm(weights.reshape(mat_d, -1), torch.t(torch.matmul(tensor1, torch.matmul(weights, torch.t(tensor2))).reshape(mat_d, -1)))
    return temp + eps * torch.eye(temp.shape[0]).cuda()

def loss_reg(weights, tensor1, tensor2, tensor3):
    temp1 = torch.matmul(weights, torch.t(tensor3).cuda()).cuda()
    temp2 = torch.matmul(tensor2, temp1.cuda()).cuda()
    temp3 = torch.matmul(tensor1, temp2.cuda().permute(1,0,2)).permute(1,0,2)
    return torch.mm(weights.reshape(1, -1), temp3.cuda().reshape(-1, 1)).reshape(1)

def weights_cat_conll2003(W1_ini,W2_ini,W3_ini):
    W1_temp = W1_ini.clone()
    W2_temp = W2_ini.clone()
    W3_temp = W3_ini.clone()
    comb_shape_0 = W1_temp.shape[0] + W2_temp.shape[0] + W3_temp.shape[0]
    comb_shape_1 = W1_temp.shape[1]
    W1 = torch.cat([W1_temp, torch.zeros(W2_temp.shape[0] + W3_temp.shape[0], comb_shape_1).cuda()], dim=0)
    W2 = torch.cat([torch.zeros(W1_temp.shape[0], comb_shape_1).cuda(), W2_temp, torch.zeros(W3_temp.shape[0], comb_shape_1).cuda()], dim=0)
    W3 = torch.cat([torch.zeros(W1_temp.shape[0] + W2_temp.shape[0], comb_shape_1).cuda(), W3_temp], dim=0)
    weights_cat_t = [W1.reshape(1, comb_shape_0, comb_shape_1), W2.reshape(1, comb_shape_0, comb_shape_1), W3.reshape(1, comb_shape_0, comb_shape_1)]
    return torch.cat(weights_cat_t, dim=0).cuda()

def calculate_reg_conll2003(W1_ini,W2_ini,W3_ini,model):
    weights = weights_cat_conll2003(W1_ini,W2_ini,W3_ini)
    print(weights.shape)
    return loss_reg(weights, model.task_covariance, model.class_covariance, model.feature_covariance)

def weights_cat_conll2000(W1_ini,W2_ini):
    W1_temp = W1_ini.clone()
    W2_temp = W2_ini.clone()
    comb_shape_0 = W1_temp.shape[0] + W2_temp.shape[0]
    comb_shape_1 = W1_temp.shape[1]
    W1 = torch.cat([W1_temp, torch.zeros(W2_temp.shape[0], comb_shape_1).cuda()], dim=0)
    W2 = torch.cat([torch.zeros(W1_temp.shape[0], comb_shape_1).cuda(), W2_temp], dim=0)
    weights_cat_t = [W1.reshape(1, comb_shape_0, comb_shape_1), W2.reshape(1, comb_shape_0, comb_shape_1)]
    return torch.cat(weights_cat_t, dim=0).cuda()

def calculate_reg_conll2000(W1_ini,W2_ini,model):
    weights = weights_cat_conll2000(W1_ini,W2_ini)
    print(weights.shape)
    return loss_reg(weights, model.task_covariance, model.class_covariance, model.feature_covariance)

def weights_cat_both_pos(W1_ini,W2_ini):
    W1_temp = W1_ini.clone()
    W2_temp = W2_ini.clone()
    comb_shape_0 = W1_temp.shape[0]
    comb_shape_1 = W1_temp.shape[1]
    W1 = torch.cat(
    [
        W1_temp[1:19,:],
        W1_temp[20:25,:],
        W1_temp[26:47,:],
        W1_temp[0,:].reshape(1,comb_shape_1),
        W1_temp[19,:].reshape(1,comb_shape_1),
        W1_temp[25,:].reshape(1,comb_shape_1)
    ],
        dim=0
    )
    W2 = torch.cat([W2_temp, torch.zeros(3, comb_shape_1).cuda()], dim=0)
    weights_cat_t = [W1.reshape(1, comb_shape_0, comb_shape_1), W2.reshape(1, comb_shape_0, comb_shape_1)]
    return torch.cat(weights_cat_t, dim=0).cuda()

def calculate_reg_both_pos(W1_ini,W2_ini,model):
    weights = weights_cat_both_pos(W1_ini,W2_ini)
    print(weights.shape)
    return loss_reg(weights, model.task_covariance, model.class_covariance, model.feature_covariance)

def train_conll2003_MRN(EPOCHS, training_loader, validation_loader, model, optimizer, checkpoint_path):
    # average validation loss
```



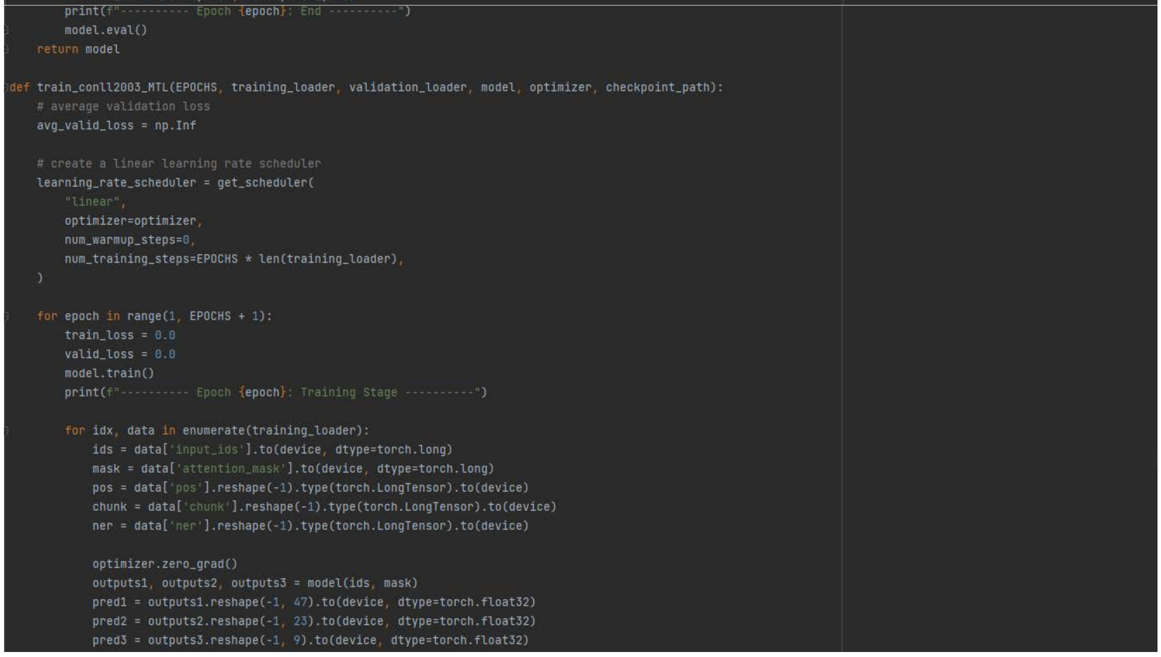
```python
avg_valid_loss = np.Inf

# create a linear learning rate scheduler
learning_rate_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=EPOCHS * len(training_loader),
)

for epoch in range(1, EPOCHS + 1):
    train_loss = 0.0
    valid_loss = 0.0
    reg_loss = 0.0
    model.train()
    print(f"---------- Epoch {epoch}: Training Stage ----------")

    for idx, data in enumerate(training_loader):
        ids = data['input_ids'].to(device, dtype=torch.long)
        mask = data['attention_mask'].to(device, dtype=torch.long)
        pos = data['pos'].reshape(-1).type(torch.LongTensor).to(device)
        chunk = data['chunk'].reshape(-1).type(torch.LongTensor).to(device)
        ner = data['ner'].reshape(-1).type(torch.LongTensor).to(device)

        optimizer.zero_grad()
        outputs1, outputs2, outputs3, W1_ini, W2_ini, W3_ini = model(ids, mask)
        # print(outputs1.shape) --> (32,256,47)
        pred1 = outputs1.reshape(-1, 47).to(device, dtype=torch.float32)
        pred2 = outputs2.reshape(-1, 23).to(device, dtype=torch.float32)
        pred3 = outputs3.reshape(-1, 9).to(device, dtype=torch.float32)

        with torch.enable_grad():
            regularizer = calculate_reg_conll2003(W1_ini, W2_ini, W3_ini, model)
            loss = loss_fn(pred1, pos) / 3 + loss_fn(pred2, chunk) / 3 + loss_fn(pred3, ner) / 3 + lambda_reg * regularizer
            loss.requires_grad_(True)
            print(f"train_loss: {loss.item()}, reg_loss: {lambda_reg * regularizer.item()}")
            loss.backward()
            optimizer.step()
            learning_rate_scheduler.step()

            train_loss += loss.item()
            reg_loss += lambda_reg * regularizer.item()

        if idx % update_frequency == 0:
            weights = weights_cat_conll2003(W1_ini, W2_ini, W3_ini)

            temp_task_covariance = covariance_mat_update(weights.data, model.class_covariance.data, model.feature_covariance.data)
            temp_class_covariance = covariance_mat_update(weights.data.permute(1, 0, 2), model.task_covariance.data, model.feature_covariance.data)
            temp_feature_covariance = covariance_mat_update(weights.data.permute(2, 0, 1), model.task_covariance.data, model.class_covariance.data)

            model.task_covariance = eig_decomp(temp_task_covariance).cuda()
            model.class_covariance = eig_decomp(temp_class_covariance).cuda()
            model.feature_covariance = eig_decomp(temp_feature_covariance).cuda()

    print(f"---------- Epoch {epoch}: Training Stage End ----------")
    train_loss = train_loss / len(training_loader)
    reg_loss = reg_loss / len(training_loader)

    print(f"---------- Epoch {epoch}: Validation Stage ----------")

    model.eval()

    with torch.no_grad():
        for idx, data in enumerate(validation_loader):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            pos = data['pos'].reshape(-1).type(torch.LongTensor).to(device)
            chunk = data['chunk'].reshape(-1).type(torch.LongTensor).to(device)
            ner = data['ner'].reshape(-1).type(torch.LongTensor).to(device)
```



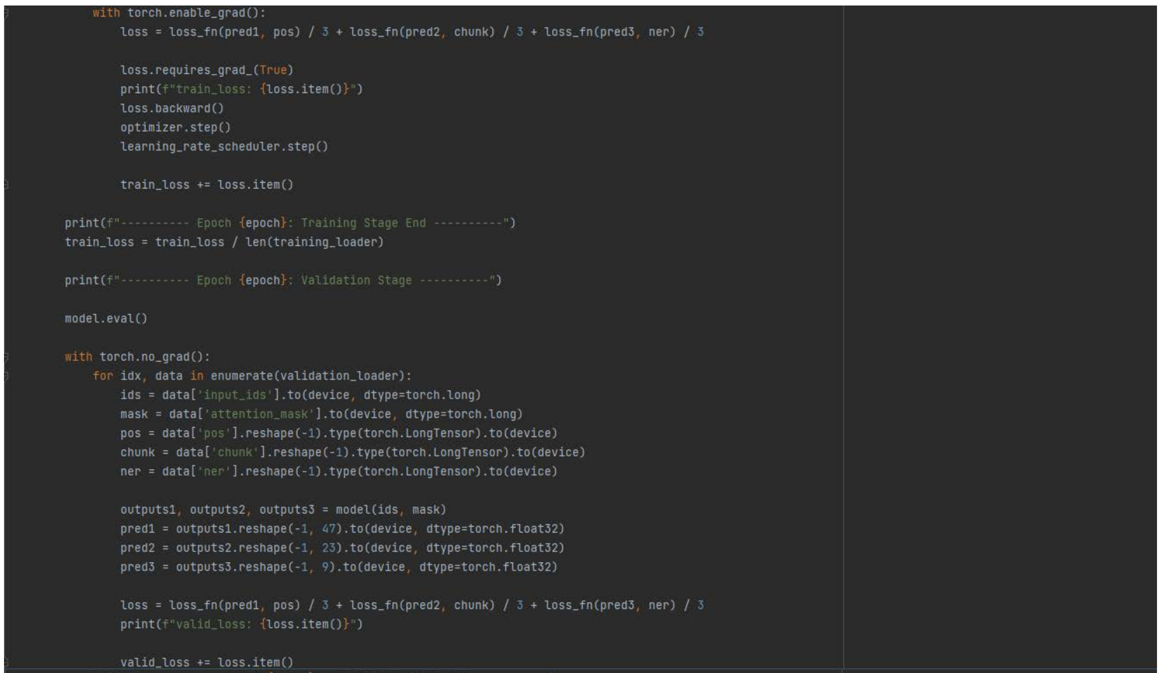
```python
                outputs1, outputs2, outputs3, W1_ini, W2_ini, W3_ini = model(ids, mask)
                pred1 = outputs1.reshape(-1, 47).to(device, dtype=torch.float32)
                pred2 = outputs2.reshape(-1, 23).to(device, dtype=torch.float32)
                pred3 = outputs3.reshape(-1, 9).to(device, dtype=torch.float32)

                regularizer = calculate_reg_conll2003(W1_ini, W2_ini, W3_ini, model)
                loss = loss_fn(pred1, pos) / 3 + loss_fn(pred2, chunk) / 3 + loss_fn(pred3, ner) / 3 + lambda_reg * regularizer
                print(f"valid_loss: {loss.item()}")

                valid_loss += loss.item()

            print(f"---------- Epoch {epoch}: Validation Stage End ----------")
            valid_loss = valid_loss / len(validation_loader)

            # Training and Validation losses
            print(f"Epoch: {epoch}, Training Loss: {train_loss}, Validation Loss: {valid_loss}")

            # checkpoint for saving
            checkpoint = {
                'epoch': epoch,
                'avg_valid_loss': valid_loss,
                'state_dict': model.state_dict(),
                'optimizer': optimizer.state_dict()
            }

            # save the checkpoint
            print("Saving the checkpoint...")
            cm.save_model(checkpoint, checkpoint_path)

            # see whether the validation loss drops or not
            if valid_loss <= avg_valid_loss:
                print(f"Validation loss decreases from {avg_valid_loss} to {valid_loss}")
                avg_valid_loss = valid_loss
            else:
                print(f"Validation loss increases from {avg_valid_loss} to {valid_loss}")
        print(f"---------- Epoch {epoch}: End ----------")

    return model

def train_conll2000_MRN(EPOCHS, training_loader, validation_loader, model, optimizer, checkpoint_path):
    # average validation loss
    avg_valid_loss = np.Inf

    # create a linear learning rate scheduler
    learning_rate_scheduler = get_scheduler(
        "linear",
        optimizer=optimizer,
        num_warmup_steps=0,
        num_training_steps=EPOCHS * len(training_loader),
    )
    for epoch in range(1, EPOCHS + 1):
        train_loss = 0.0
        valid_loss = 0.0
        reg_loss = 0.0

        model.train()
        print(f"---------- Epoch {epoch}: Training Stage ----------")

        for idx, data in enumerate(training_loader):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            pos = data['pos'].reshape(-1).type(torch.LongTensor).to(device)
            chunk = data['chunk'].reshape(-1).type(torch.LongTensor).to(device)

            optimizer.zero_grad()
            outputs1, outputs2, W1_ini, W2_ini = model(ids, mask)

            pred1 = outputs1.reshape(-1, 44).to(device, dtype=torch.float32)
            pred2 = outputs2.reshape(-1, 23).to(device, dtype=torch.float32)
```



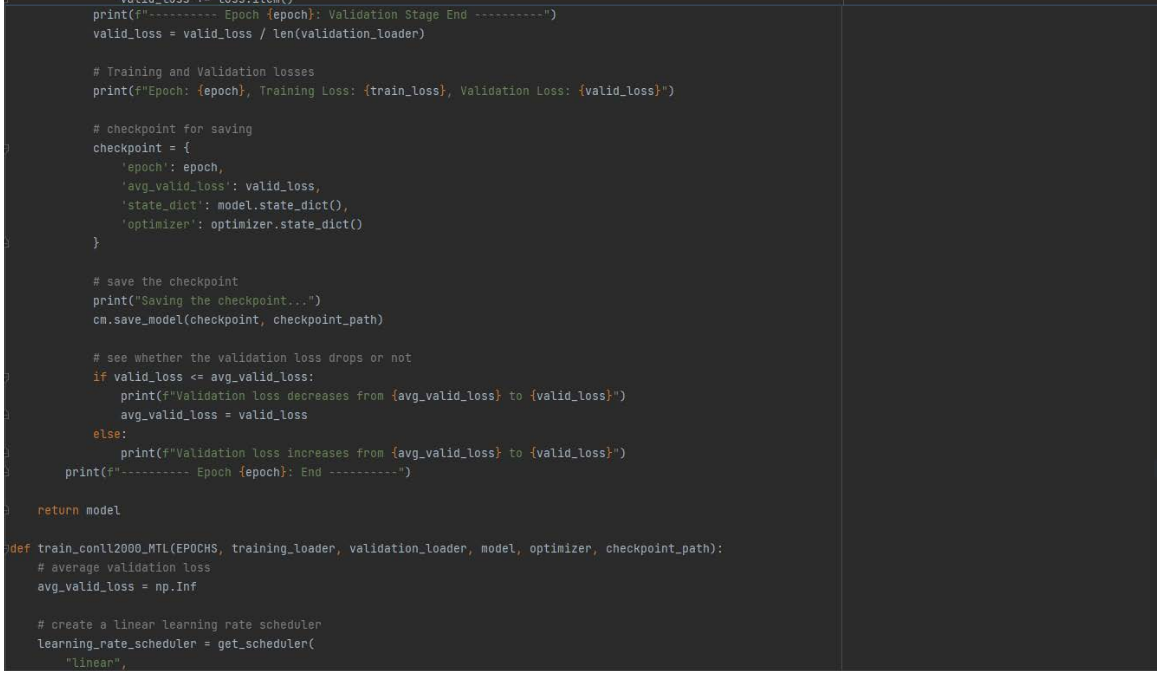
```
            with torch.enable_grad():
                regularizer = calculate_reg_conll2000(W1_ini, W2_ini, model)
                loss = loss_fn(pred1, pos) / 2 + loss_fn(pred2, chunk) / 2 + lambda_reg * regularizer

                loss.requires_grad_(True)
                print(f"train_loss: {loss.item()}, reg_loss: {lambda_reg * regularizer.item()}")
                loss.backward()
                optimizer.step()
                learning_rate_scheduler.step()

                train_loss += loss.item()
                reg_loss += lambda_reg * regularizer.item()

            if idx % update_frequency == 0:
                weights = weights_cat_conll2000(W1_ini, W2_ini)

                temp_task_covariance = covariance_mat_update(weights.data, model.class_covariance.data, model.feature_covariance.data)
                temp_class_covariance = covariance_mat_update(weights.data.permute(1, 0, 2), model.task_covariance.data, model.feature_covariance.data)
                temp_feature_covariance = covariance_mat_update(weights.data.permute(2, 0, 1), model.task_covariance.data, model.class_covariance.data)

                model.task_covariance = eig_decomp(temp_task_covariance).cuda()
                model.class_covariance = eig_decomp(temp_class_covariance).cuda()
                model.feature_covariance = eig_decomp(temp_feature_covariance).cuda()

        print(f"---------- Epoch {epoch}: Training Stage End ----------")
        train_loss = train_loss / len(training_loader)
        reg_loss = reg_loss / len(training_loader)

        checkpoint = {
            'epoch': epoch,
            'avg_valid_loss': train_loss,
            'state_dict': model.state_dict(),
            'optimizer': optimizer.state_dict()
        }
        cm.save_model(checkpoint, checkpoint_path)
```



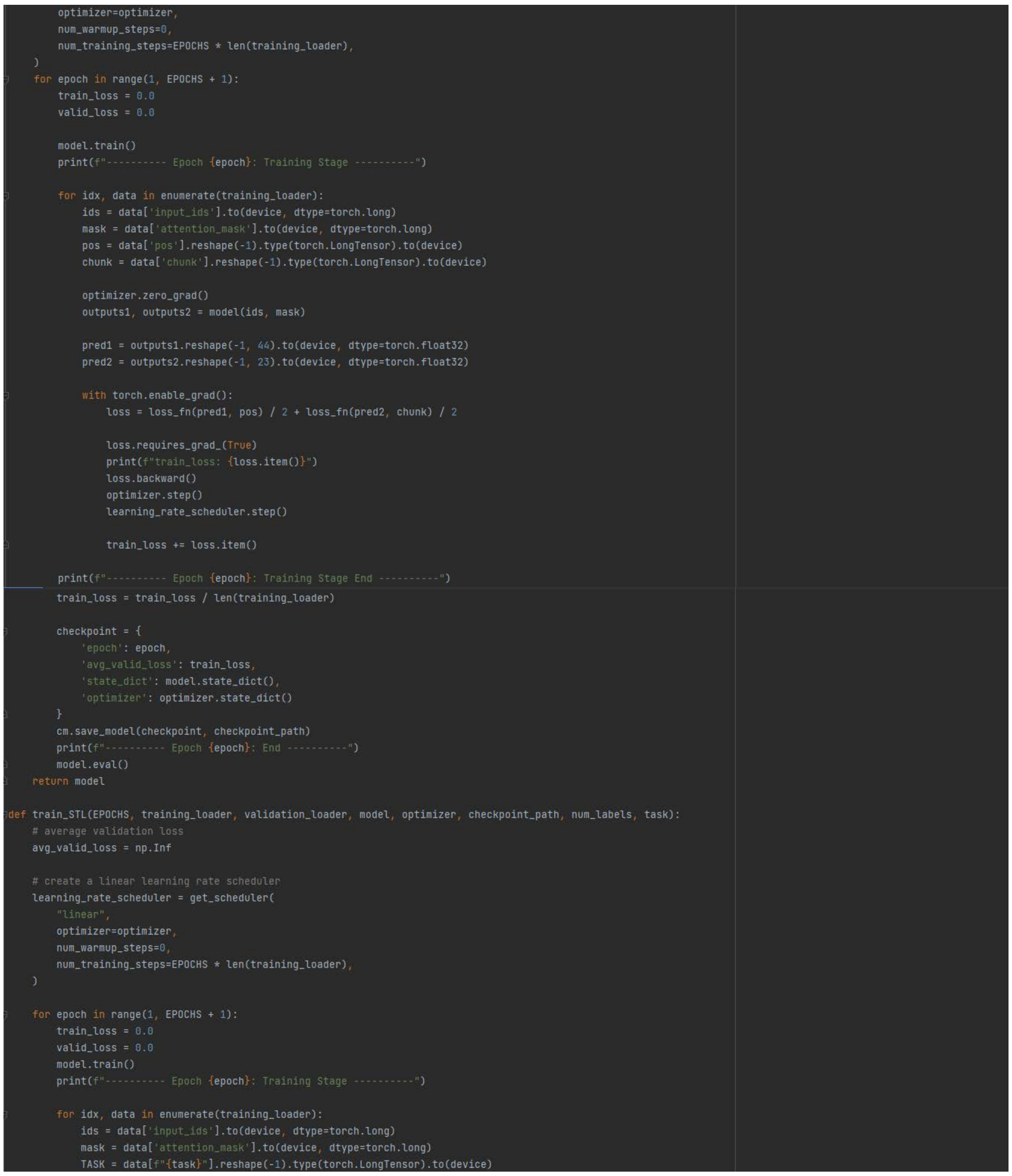
```
        print(f"---------- Epoch {epoch}: End ----------")
        model.eval()
    return model

def train_conll2003_MTL(EPOCHS, training_loader, validation_loader, model, optimizer, checkpoint_path):
    # average validation loss
    avg_valid_loss = np.Inf

    # create a linear learning rate scheduler
    learning_rate_scheduler = get_scheduler(
        "linear",
        optimizer=optimizer,
        num_warmup_steps=0,
        num_training_steps=EPOCHS * len(training_loader),
    )

    for epoch in range(1, EPOCHS + 1):
        train_loss = 0.0
        valid_loss = 0.0
        model.train()
        print(f"---------- Epoch {epoch}: Training Stage ----------")

        for idx, data in enumerate(training_loader):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            pos = data['pos'].reshape(-1).type(torch.LongTensor).to(device)
            chunk = data['chunk'].reshape(-1).type(torch.LongTensor).to(device)
            ner = data['ner'].reshape(-1).type(torch.LongTensor).to(device)

            optimizer.zero_grad()
            outputs1, outputs2, outputs3 = model(ids, mask)
            pred1 = outputs1.reshape(-1, 47).to(device, dtype=torch.float32)
            pred2 = outputs2.reshape(-1, 23).to(device, dtype=torch.float32)
            pred3 = outputs3.reshape(-1, 9).to(device, dtype=torch.float32)
```



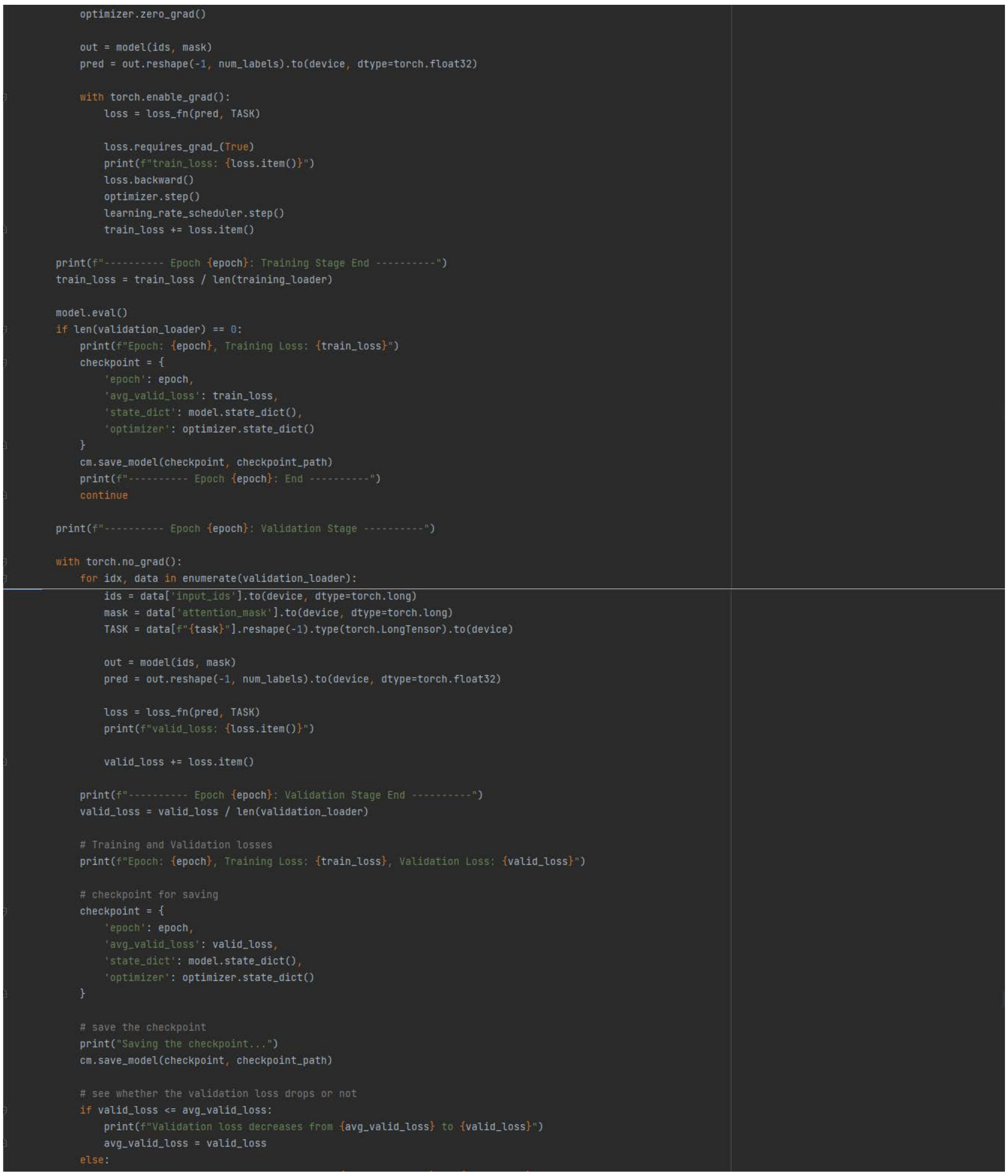
```
            with torch.enable_grad():
                loss = loss_fn(pred1, pos) / 3 + loss_fn(pred2, chunk) / 3 + loss_fn(pred3, ner) / 3

                loss.requires_grad_(True)
                print(f"train_loss: {loss.item()}")
                loss.backward()
                optimizer.step()
                learning_rate_scheduler.step()

                train_loss += loss.item()

        print(f"---------- Epoch {epoch}: Training Stage End ----------")
        train_loss = train_loss / len(training_loader)

        print(f"---------- Epoch {epoch}: Validation Stage ----------")

        model.eval()

        with torch.no_grad():
            for idx, data in enumerate(validation_loader):
                ids = data['input_ids'].to(device, dtype=torch.long)
                mask = data['attention_mask'].to(device, dtype=torch.long)
                pos = data['pos'].reshape(-1).type(torch.LongTensor).to(device)
                chunk = data['chunk'].reshape(-1).type(torch.LongTensor).to(device)
                ner = data['ner'].reshape(-1).type(torch.LongTensor).to(device)

                outputs1, outputs2, outputs3 = model(ids, mask)
                pred1 = outputs1.reshape(-1, 47).to(device, dtype=torch.float32)
                pred2 = outputs2.reshape(-1, 23).to(device, dtype=torch.float32)
                pred3 = outputs3.reshape(-1, 9).to(device, dtype=torch.float32)

                loss = loss_fn(pred1, pos) / 3 + loss_fn(pred2, chunk) / 3 + loss_fn(pred3, ner) / 3
                print(f"valid_loss: {loss.item()}")

                valid_loss += loss.item()
```



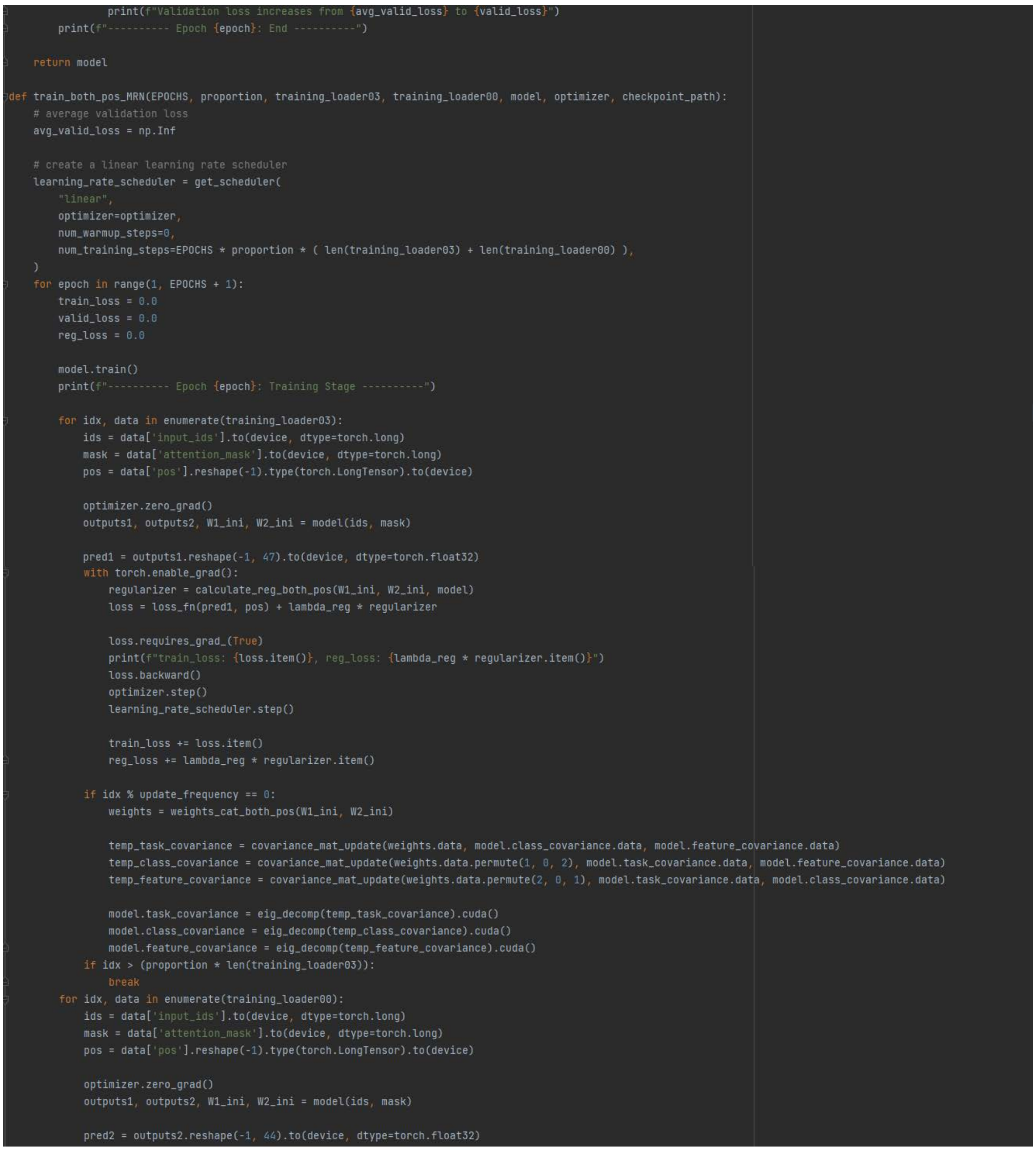
```
            print(f"---------- Epoch {epoch}: Validation Stage End ----------")
            valid_loss = valid_loss / len(validation_loader)

            # Training and Validation losses
            print(f"Epoch: {epoch}, Training Loss: {train_loss}, Validation Loss: {valid_loss}")

            # checkpoint for saving
            checkpoint = {
                'epoch': epoch,
                'avg_valid_loss': valid_loss,
                'state_dict': model.state_dict(),
                'optimizer': optimizer.state_dict()
            }

            # save the checkpoint
            print("Saving the checkpoint...")
            cm.save_model(checkpoint, checkpoint_path)

            # see whether the validation loss drops or not
            if valid_loss <= avg_valid_loss:
                print(f"Validation loss decreases from {avg_valid_loss} to {valid_loss}")
                avg_valid_loss = valid_loss
            else:
                print(f"Validation loss increases from {avg_valid_loss} to {valid_loss}")
        print(f"---------- Epoch {epoch}: End ----------")

    return model

def train_conll2000_MTL(EPOCHS, training_loader, validation_loader, model, optimizer, checkpoint_path):
    # average validation loss
    avg_valid_loss = np.Inf

    # create a linear learning rate scheduler
    learning_rate_scheduler = get_scheduler(
        "linear",
```



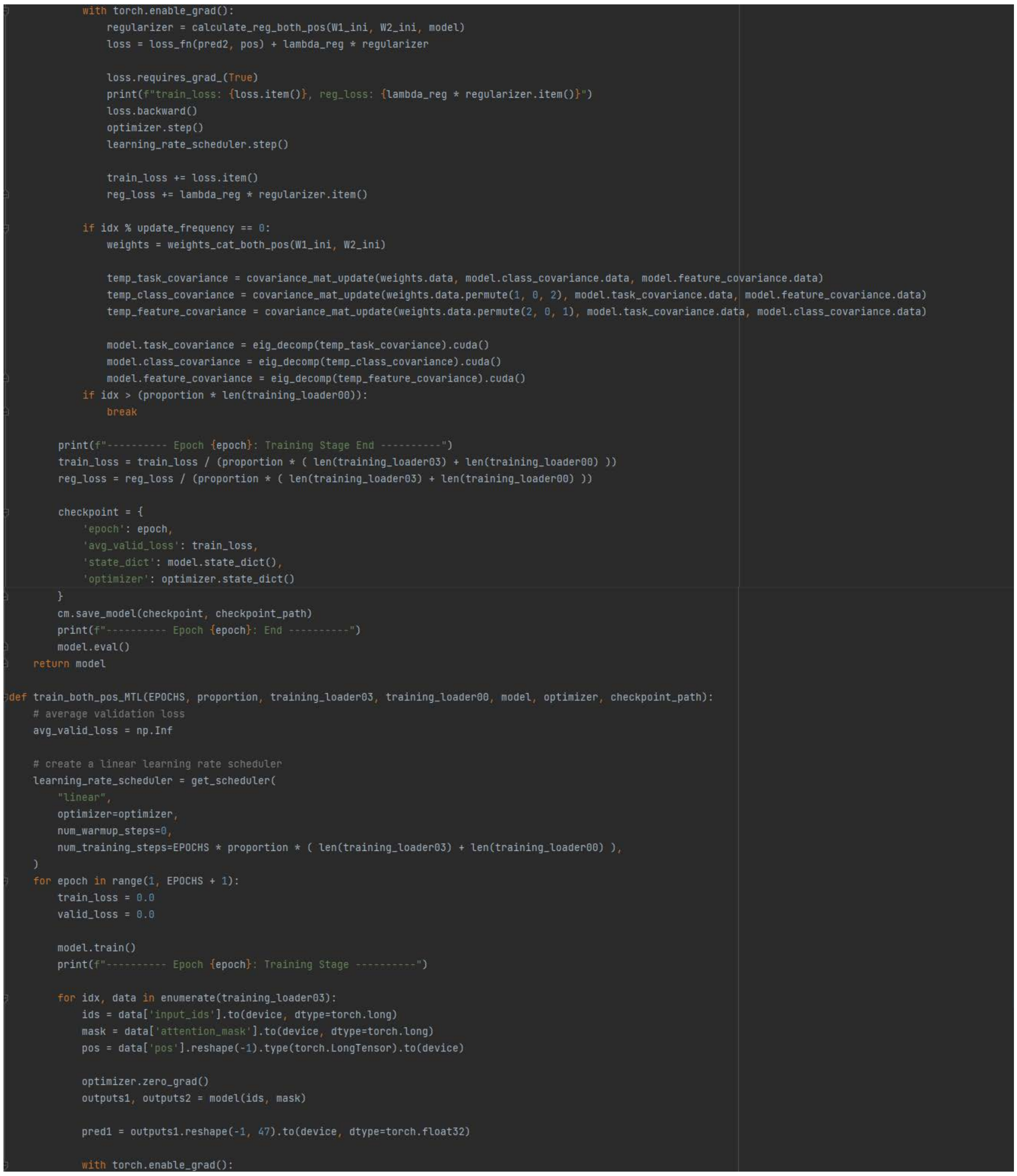
```
        optimizer=optimizer,
        num_warmup_steps=0,
        num_training_steps=EPOCHS * len(training_loader),
    )
    for epoch in range(1, EPOCHS + 1):
        train_loss = 0.0
        valid_loss = 0.0

        model.train()
        print(f"--------- Epoch {epoch}: Training Stage ---------")

        for idx, data in enumerate(training_loader):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            pos = data['pos'].reshape(-1).type(torch.LongTensor).to(device)
            chunk = data['chunk'].reshape(-1).type(torch.LongTensor).to(device)

            optimizer.zero_grad()
            outputs1, outputs2 = model(ids, mask)

            pred1 = outputs1.reshape(-1, 44).to(device, dtype=torch.float32)
            pred2 = outputs2.reshape(-1, 23).to(device, dtype=torch.float32)

            with torch.enable_grad():
                loss = loss_fn(pred1, pos) / 2 + loss_fn(pred2, chunk) / 2

                loss.requires_grad_(True)
                print(f"train_loss: {loss.item()}")
                loss.backward()
                optimizer.step()
                learning_rate_scheduler.step()

                train_loss += loss.item()

        print(f"--------- Epoch {epoch}: Training Stage End ---------")
        train_loss = train_loss / len(training_loader)

        checkpoint = {
            'epoch': epoch,
            'avg_valid_loss': train_loss,
            'state_dict': model.state_dict(),
            'optimizer': optimizer.state_dict()
        }
        cm.save_model(checkpoint, checkpoint_path)
        print(f"--------- Epoch {epoch}: End ---------")
        model.eval()
    return model

def train_STL(EPOCHS, training_loader, validation_loader, model, optimizer, checkpoint_path, num_labels, task):
    # average validation loss
    avg_valid_loss = np.Inf

    # create a linear learning rate scheduler
    learning_rate_scheduler = get_scheduler(
        "linear",
        optimizer=optimizer,
        num_warmup_steps=0,
        num_training_steps=EPOCHS * len(training_loader),
    )

    for epoch in range(1, EPOCHS + 1):
        train_loss = 0.0
        valid_loss = 0.0
        model.train()
        print(f"--------- Epoch {epoch}: Training Stage ---------")

        for idx, data in enumerate(training_loader):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            TASK = data[f"{task}"].reshape(-1).type(torch.LongTensor).to(device)
```



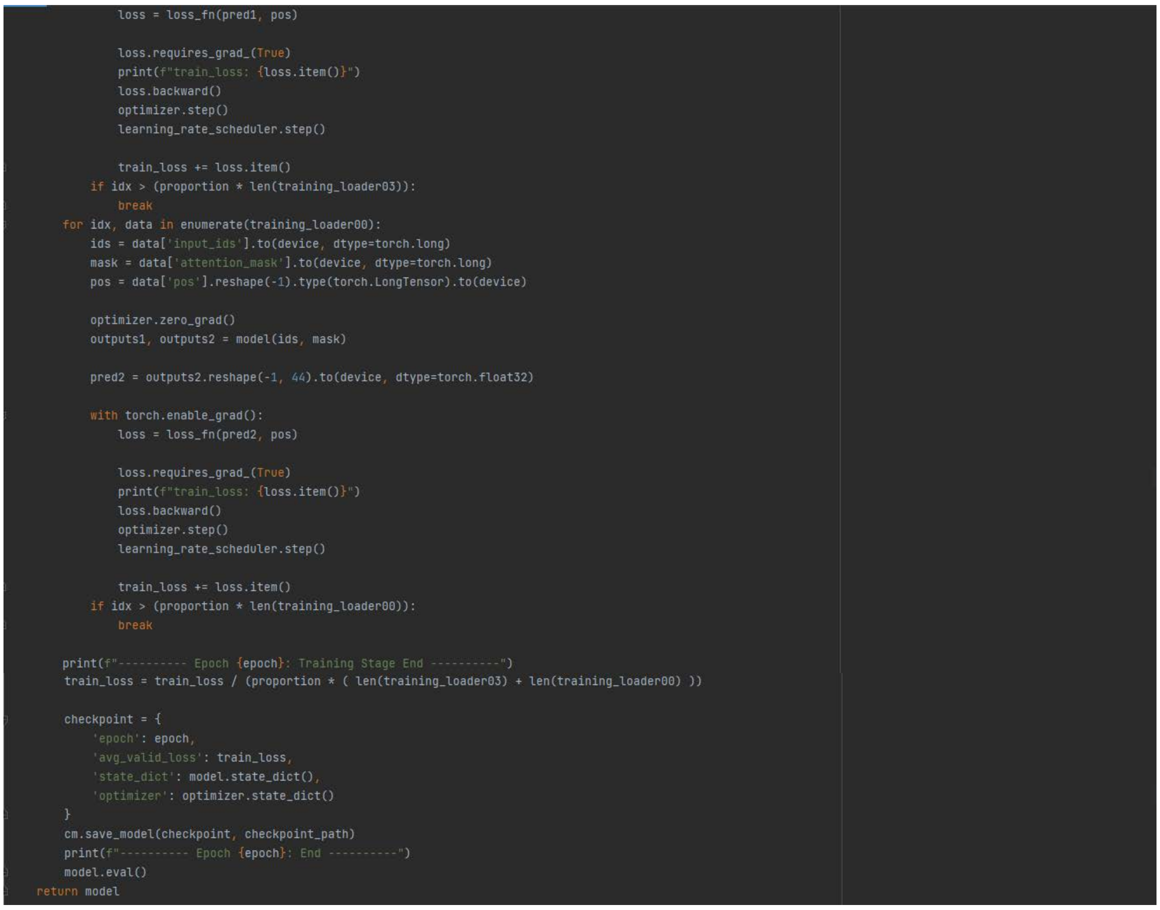
```
            optimizer.zero_grad()

            out = model(ids, mask)
            pred = out.reshape(-1, num_labels).to(device, dtype=torch.float32)

            with torch.enable_grad():
                loss = loss_fn(pred, TASK)

                loss.requires_grad_(True)
                print(f"train_loss: {loss.item()}")
                loss.backward()
                optimizer.step()
                learning_rate_scheduler.step()
                train_loss += loss.item()

        print(f"---------- Epoch {epoch}: Training Stage End ----------")
        train_loss = train_loss / len(training_loader)

        model.eval()
        if len(validation_loader) == 0:
            print(f"Epoch: {epoch}, Training Loss: {train_loss}")
            checkpoint = {
                'epoch': epoch,
                'avg_valid_loss': train_loss,
                'state_dict': model.state_dict(),
                'optimizer': optimizer.state_dict()
            }
            cm.save_model(checkpoint, checkpoint_path)
            print(f"---------- Epoch {epoch}: End ----------")
            continue

        print(f"---------- Epoch {epoch}: Validation Stage ----------")

        with torch.no_grad():
            for idx, data in enumerate(validation_loader):
                ids = data['input_ids'].to(device, dtype=torch.long)
                mask = data['attention_mask'].to(device, dtype=torch.long)
                TASK = data[f"{task}"].reshape(-1).type(torch.LongTensor).to(device)

                out = model(ids, mask)
                pred = out.reshape(-1, num_labels).to(device, dtype=torch.float32)

                loss = loss_fn(pred, TASK)
                print(f"valid_loss: {loss.item()}")

                valid_loss += loss.item()

            print(f"---------- Epoch {epoch}: Validation Stage End ----------")
            valid_loss = valid_loss / len(validation_loader)

            # Training and Validation losses
            print(f"Epoch: {epoch}, Training Loss: {train_loss}, Validation Loss: {valid_loss}")

            # checkpoint for saving
            checkpoint = {
                'epoch': epoch,
                'avg_valid_loss': valid_loss,
                'state_dict': model.state_dict(),
                'optimizer': optimizer.state_dict()
            }

            # save the checkpoint
            print("Saving the checkpoint...")
            cm.save_model(checkpoint, checkpoint_path)

            # see whether the validation loss drops or not
            if valid_loss <= avg_valid_loss:
                print(f"Validation loss decreases from {avg_valid_loss} to {valid_loss}")
                avg_valid_loss = valid_loss
            else:
```



```
                print(f"Validation loss increases from {avg_valid_loss} to {valid_loss}")
        print(f"---------- Epoch {epoch}: End ----------")

    return model

def train_both_pos_MRN(EPOCHS, proportion, training_loader03, training_loader00, model, optimizer, checkpoint_path):
    # average validation loss
    avg_valid_loss = np.Inf

    # create a linear learning rate scheduler
    learning_rate_scheduler = get_scheduler(
        "linear",
        optimizer=optimizer,
        num_warmup_steps=0,
        num_training_steps=EPOCHS * proportion * ( len(training_loader03) + len(training_loader00) ),
    )
    for epoch in range(1, EPOCHS + 1):
        train_loss = 0.0
        valid_loss = 0.0
        reg_loss = 0.0

        model.train()
        print(f"---------- Epoch {epoch}: Training Stage ----------")

        for idx, data in enumerate(training_loader03):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            pos = data['pos'].reshape(-1).type(torch.LongTensor).to(device)

            optimizer.zero_grad()
            outputs1, outputs2, W1_ini, W2_ini = model(ids, mask)

            pred1 = outputs1.reshape(-1, 47).to(device, dtype=torch.float32)
            with torch.enable_grad():
                regularizer = calculate_reg_both_pos(W1_ini, W2_ini, model)
                loss = loss_fn(pred1, pos) + lambda_reg * regularizer

                loss.requires_grad_(True)
                print(f"train_loss: {loss.item()}, reg_loss: {lambda_reg * regularizer.item()}")
                loss.backward()
                optimizer.step()
                learning_rate_scheduler.step()

                train_loss += loss.item()
                reg_loss += lambda_reg * regularizer.item()

            if idx % update_frequency == 0:
                weights = weights_cat_both_pos(W1_ini, W2_ini)

                temp_task_covariance = covariance_mat_update(weights.data, model.class_covariance.data, model.feature_covariance.data)
                temp_class_covariance = covariance_mat_update(weights.data.permute(1, 0, 2), model.task_covariance.data, model.feature_covariance.data)
                temp_feature_covariance = covariance_mat_update(weights.data.permute(2, 0, 1), model.task_covariance.data, model.class_covariance.data)

                model.task_covariance = eig_decomp(temp_task_covariance).cuda()
                model.class_covariance = eig_decomp(temp_class_covariance).cuda()
                model.feature_covariance = eig_decomp(temp_feature_covariance).cuda()
            if idx > (proportion * len(training_loader03)):
                break
        for idx, data in enumerate(training_loader00):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            pos = data['pos'].reshape(-1).type(torch.LongTensor).to(device)

            optimizer.zero_grad()
            outputs1, outputs2, W1_ini, W2_ini = model(ids, mask)

            pred2 = outputs2.reshape(-1, 44).to(device, dtype=torch.float32)
```

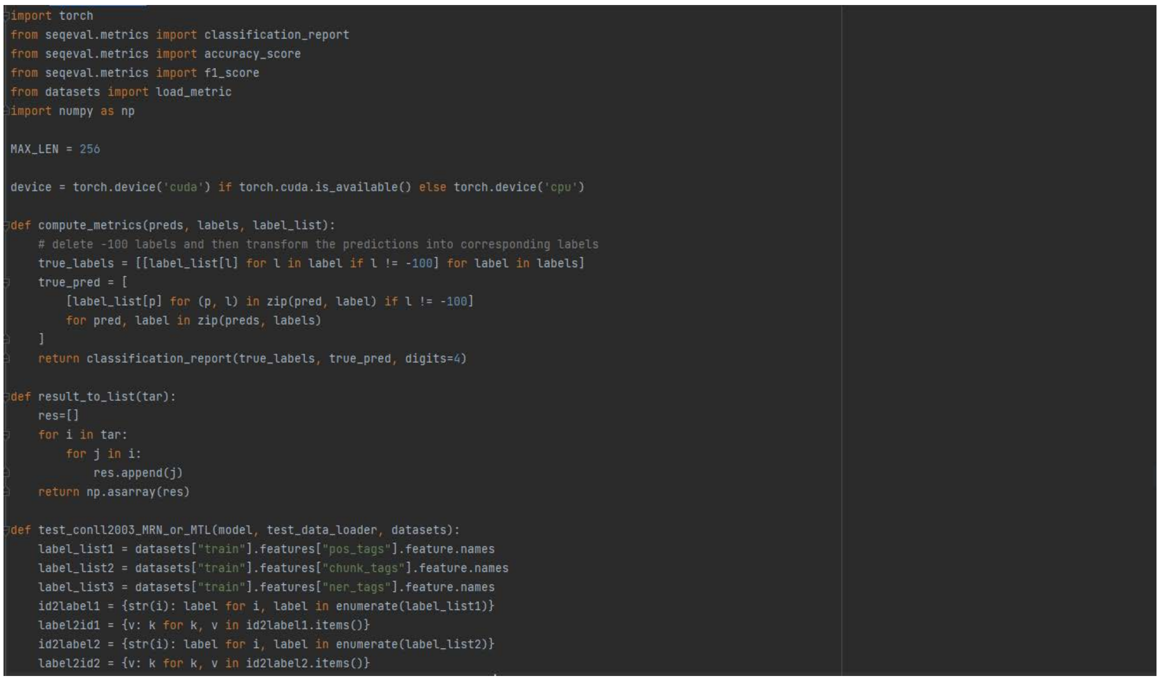


```python
            with torch.enable_grad():
                regularizer = calculate_reg_both_pos(W1_ini, W2_ini, model)
                loss = loss_fn(pred2, pos) + lambda_reg * regularizer

                loss.requires_grad_(True)
                print(f"train_loss: {loss.item()}, reg_loss: {lambda_reg * regularizer.item()}")
                loss.backward()
                optimizer.step()
                learning_rate_scheduler.step()

                train_loss += loss.item()
                reg_loss += lambda_reg * regularizer.item()

            if idx % update_frequency == 0:
                weights = weights_cat_both_pos(W1_ini, W2_ini)

                temp_task_covariance = covariance_mat_update(weights.data, model.class_covariance.data, model.feature_covariance.data)
                temp_class_covariance = covariance_mat_update(weights.data.permute(1, 0, 2), model.task_covariance.data, model.feature_covariance.data)
                temp_feature_covariance = covariance_mat_update(weights.data.permute(2, 0, 1), model.task_covariance.data, model.class_covariance.data)

                model.task_covariance = eig_decomp(temp_task_covariance).cuda()
                model.class_covariance = eig_decomp(temp_class_covariance).cuda()
                model.feature_covariance = eig_decomp(temp_feature_covariance).cuda()
            if idx > (proportion * len(training_loader00)):
                break

        print(f"---------- Epoch {epoch}: Training Stage End ----------")
        train_loss = train_loss / (proportion * ( len(training_loader03) + len(training_loader00) ))
        reg_loss = reg_loss / (proportion * ( len(training_loader03) + len(training_loader00) ))

        checkpoint = {
            'epoch': epoch,
            'avg_valid_loss': train_loss,
            'state_dict': model.state_dict(),
            'optimizer': optimizer.state_dict()
        }
        cm.save_model(checkpoint, checkpoint_path)
        print(f"---------- Epoch {epoch}: End ----------")
        model.eval()
    return model

def train_both_pos_MTL(EPOCHS, proportion, training_loader03, training_loader00, model, optimizer, checkpoint_path):
    # average validation loss
    avg_valid_loss = np.Inf

    # create a linear learning rate scheduler
    learning_rate_scheduler = get_scheduler(
        "linear",
        optimizer=optimizer,
        num_warmup_steps=0,
        num_training_steps=EPOCHS * proportion * ( len(training_loader03) + len(training_loader00) ),
    )
    for epoch in range(1, EPOCHS + 1):
        train_loss = 0.0
        valid_loss = 0.0

        model.train()
        print(f"---------- Epoch {epoch}: Training Stage ----------")

        for idx, data in enumerate(training_loader03):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            pos = data['pos'].reshape(-1).type(torch.LongTensor).to(device)

            optimizer.zero_grad()
            outputs1, outputs2 = model(ids, mask)

            pred1 = outputs1.reshape(-1, 47).to(device, dtype=torch.float32)

            with torch.enable_grad():
```

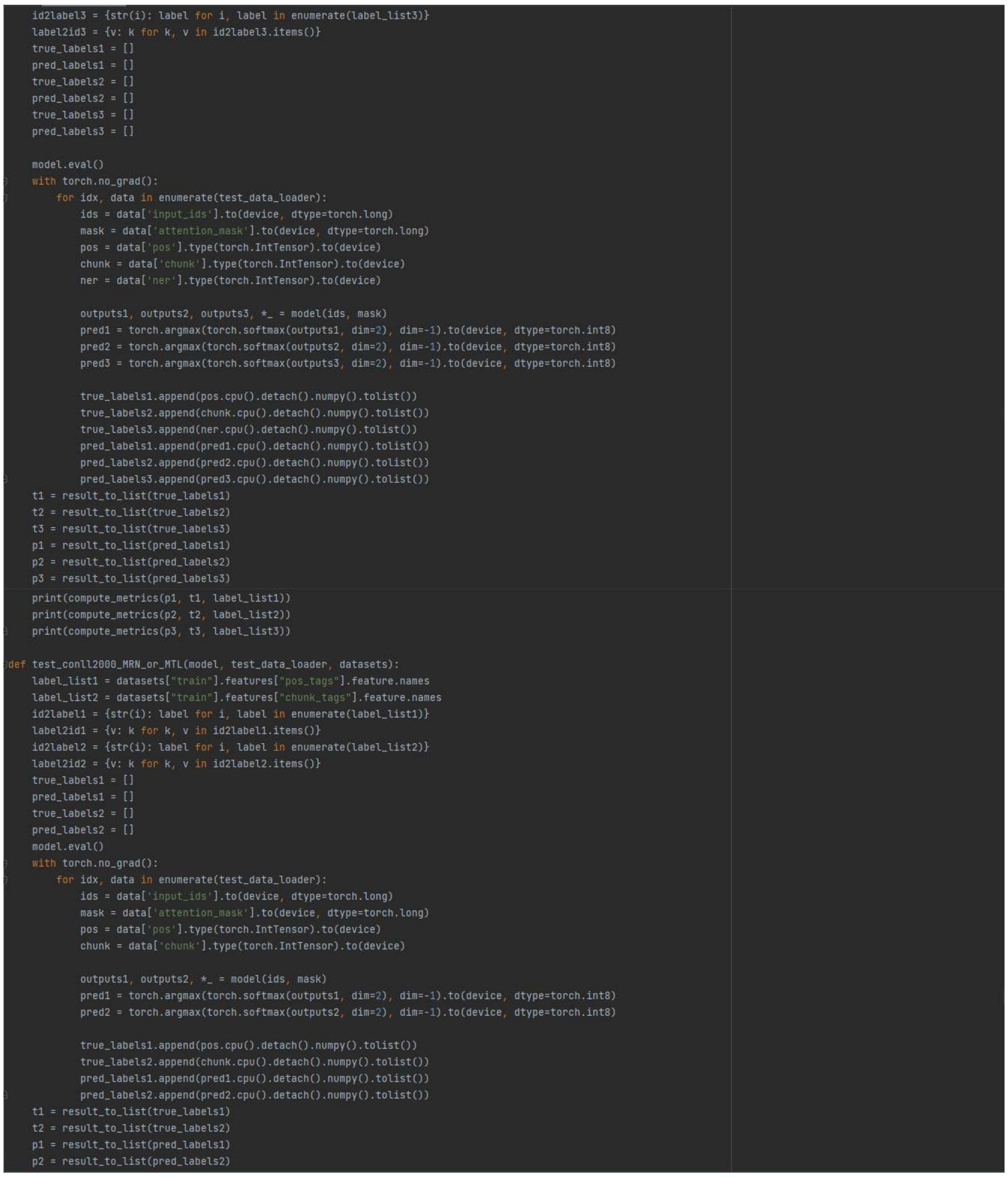


```python
                loss = loss_fn(pred1, pos)

                loss.requires_grad_(True)
                print(f"train_loss: {loss.item()}")
                loss.backward()
                optimizer.step()
                learning_rate_scheduler.step()

                train_loss += loss.item()
            if idx > (proportion * len(training_loader03)):
                break
        for idx, data in enumerate(training_loader00):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            pos = data['pos'].reshape(-1).type(torch.LongTensor).to(device)

            optimizer.zero_grad()
            outputs1, outputs2 = model(ids, mask)

            pred2 = outputs2.reshape(-1, 44).to(device, dtype=torch.float32)

            with torch.enable_grad():
                loss = loss_fn(pred2, pos)

                loss.requires_grad_(True)
                print(f"train_loss: {loss.item()}")
                loss.backward()
                optimizer.step()
                learning_rate_scheduler.step()

                train_loss += loss.item()
            if idx > (proportion * len(training_loader00)):
                break

        print(f"---------- Epoch {epoch}: Training Stage End ----------")
        train_loss = train_loss / (proportion * ( len(training_loader03) + len(training_loader00) ))

        checkpoint = {
            'epoch': epoch,
            'avg_valid_loss': train_loss,
            'state_dict': model.state_dict(),
            'optimizer': optimizer.state_dict()
        }
        cm.save_model(checkpoint, checkpoint_path)
        print(f"---------- Epoch {epoch}: End ----------")
        model.eval()
    return model
```

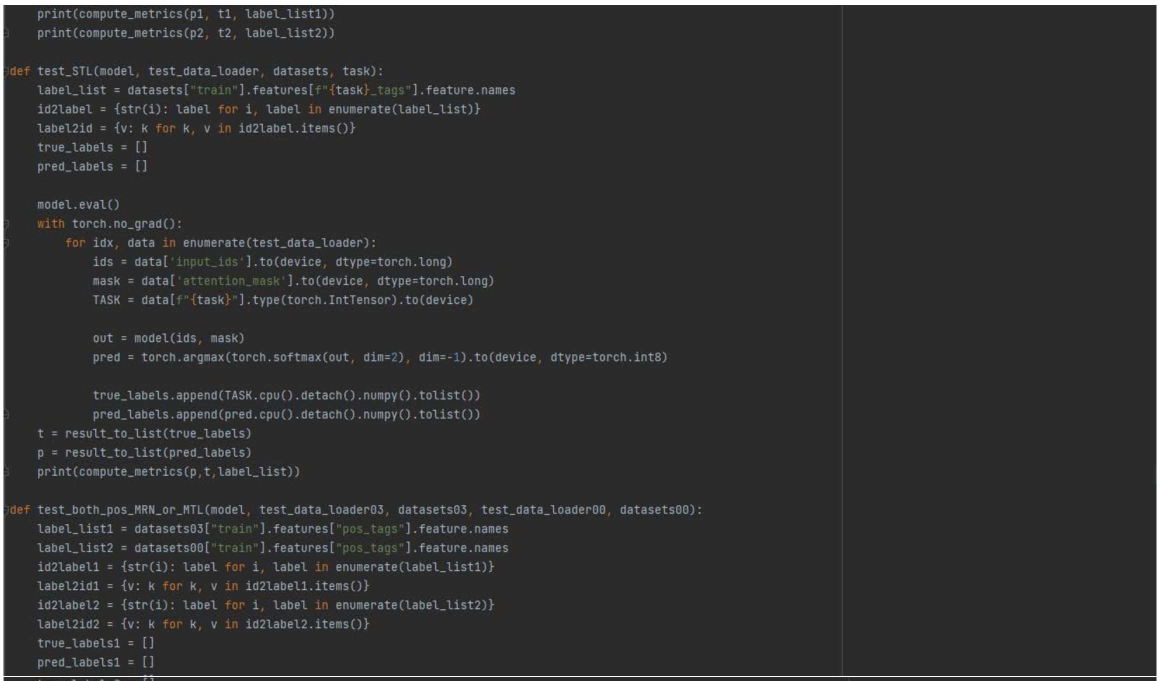
A.5: trainers.py



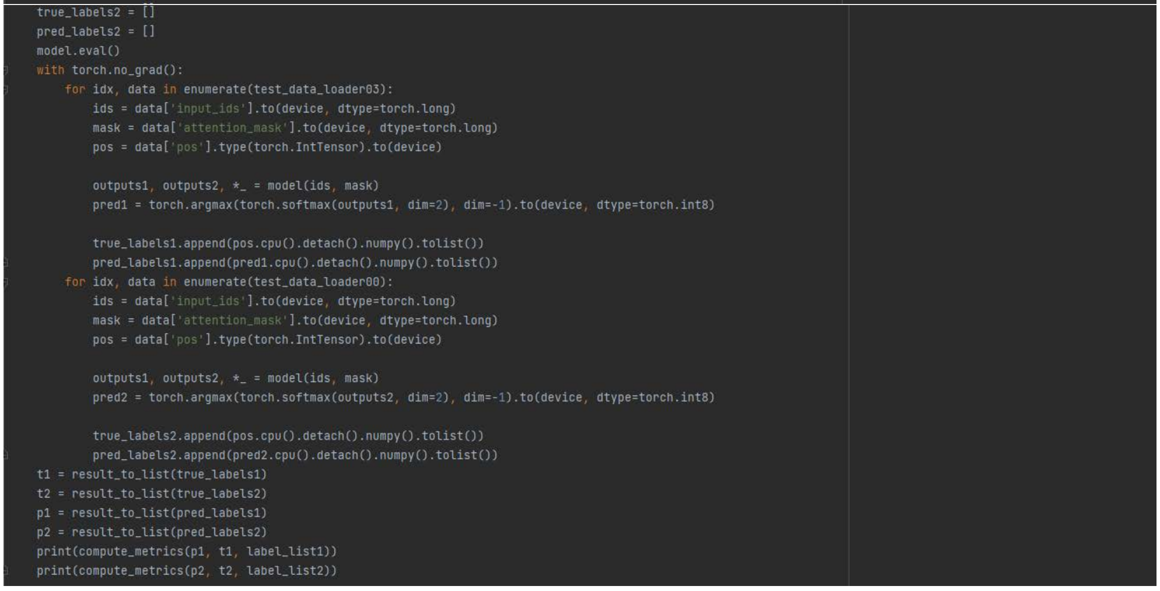
```python
import torch
from seqeval.metrics import classification_report
from seqeval.metrics import accuracy_score
from seqeval.metrics import f1_score
from datasets import load_metric
import numpy as np

MAX_LEN = 256

device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

def compute_metrics(preds, labels, label_list):
    # delete -100 labels and then transform the predictions into corresponding labels
    true_labels = [[label_list[l] for l in label if l != -100] for label in labels]
    true_pred = [
        [label_list[p] for (p, l) in zip(pred, label) if l != -100]
        for pred, label in zip(preds, labels)
    ]
    return classification_report(true_labels, true_pred, digits=4)

def result_to_list(tar):
    res=[]
    for i in tar:
        for j in i:
            res.append(j)
    return np.asarray(res)

def test_conll2003_MRN_or_MTL(model, test_data_loader, datasets):
    label_list1 = datasets["train"].features["pos_tags"].feature.names
    label_list2 = datasets["train"].features["chunk_tags"].feature.names
    label_list3 = datasets["train"].features["ner_tags"].feature.names
    id2label1 = {str(i): label for i, label in enumerate(label_list1)}
    label2id1 = {v: k for k, v in id2label1.items()}
    id2label2 = {str(i): label for i, label in enumerate(label_list2)}
    label2id2 = {v: k for k, v in id2label2.items()}
```



```python
    id2label3 = {str(i): label for i, label in enumerate(label_list3)}
    label2id3 = {v: k for k, v in id2label3.items()}
    true_labels1 = []
    pred_labels1 = []
    true_labels2 = []
    pred_labels2 = []
    true_labels3 = []
    pred_labels3 = []

    model.eval()
    with torch.no_grad():
        for idx, data in enumerate(test_data_loader):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            pos = data['pos'].type(torch.IntTensor).to(device)
            chunk = data['chunk'].type(torch.IntTensor).to(device)
            ner = data['ner'].type(torch.IntTensor).to(device)

            outputs1, outputs2, outputs3, *_ = model(ids, mask)
            pred1 = torch.argmax(torch.softmax(outputs1, dim=2), dim=-1).to(device, dtype=torch.int8)
            pred2 = torch.argmax(torch.softmax(outputs2, dim=2), dim=-1).to(device, dtype=torch.int8)
            pred3 = torch.argmax(torch.softmax(outputs3, dim=2), dim=-1).to(device, dtype=torch.int8)

            true_labels1.append(pos.cpu().detach().numpy().tolist())
            true_labels2.append(chunk.cpu().detach().numpy().tolist())
            true_labels3.append(ner.cpu().detach().numpy().tolist())
            pred_labels1.append(pred1.cpu().detach().numpy().tolist())
            pred_labels2.append(pred2.cpu().detach().numpy().tolist())
            pred_labels3.append(pred3.cpu().detach().numpy().tolist())
    t1 = result_to_list(true_labels1)
    t2 = result_to_list(true_labels2)
    t3 = result_to_list(true_labels3)
    p1 = result_to_list(pred_labels1)
    p2 = result_to_list(pred_labels2)
    p3 = result_to_list(pred_labels3)
    print(compute_metrics(p1, t1, label_list1))
    print(compute_metrics(p2, t2, label_list2))
    print(compute_metrics(p3, t3, label_list3))

def test_conll2000_MRN_or_MTL(model, test_data_loader, datasets):
    label_list1 = datasets["train"].features["pos_tags"].feature.names
    label_list2 = datasets["train"].features["chunk_tags"].feature.names
    id2label1 = {str(i): label for i, label in enumerate(label_list1)}
    label2id1 = {v: k for k, v in id2label1.items()}
    id2label2 = {str(i): label for i, label in enumerate(label_list2)}
    label2id2 = {v: k for k, v in id2label2.items()}
    true_labels1 = []
    pred_labels1 = []
    true_labels2 = []
    pred_labels2 = []
    model.eval()
    with torch.no_grad():
        for idx, data in enumerate(test_data_loader):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            pos = data['pos'].type(torch.IntTensor).to(device)
            chunk = data['chunk'].type(torch.IntTensor).to(device)

            outputs1, outputs2, *_ = model(ids, mask)
            pred1 = torch.argmax(torch.softmax(outputs1, dim=2), dim=-1).to(device, dtype=torch.int8)
            pred2 = torch.argmax(torch.softmax(outputs2, dim=2), dim=-1).to(device, dtype=torch.int8)

            true_labels1.append(pos.cpu().detach().numpy().tolist())
            true_labels2.append(chunk.cpu().detach().numpy().tolist())
            pred_labels1.append(pred1.cpu().detach().numpy().tolist())
            pred_labels2.append(pred2.cpu().detach().numpy().tolist())
    t1 = result_to_list(true_labels1)
    t2 = result_to_list(true_labels2)
    p1 = result_to_list(pred_labels1)
    p2 = result_to_list(pred_labels2)
```



```
    print(compute_metrics(p1, t1, label_list1))
    print(compute_metrics(p2, t2, label_list2))

def test_STL(model, test_data_loader, datasets, task):
    label_list = datasets["train"].features[f"{task}_tags"].feature.names
    id2label = {str(i): label for i, label in enumerate(label_list)}
    label2id = {v: k for k, v in id2label.items()}
    true_labels = []
    pred_labels = []

    model.eval()
    with torch.no_grad():
        for idx, data in enumerate(test_data_loader):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            TASK = data[f"{task}"].type(torch.IntTensor).to(device)

            out = model(ids, mask)
            pred = torch.argmax(torch.softmax(out, dim=2), dim=-1).to(device, dtype=torch.int8)

            true_labels.append(TASK.cpu().detach().numpy().tolist())
            pred_labels.append(pred.cpu().detach().numpy().tolist())
    t = result_to_list(true_labels)
    p = result_to_list(pred_labels)
    print(compute_metrics(p,t,label_list))

def test_both_pos_MRN_or_MTL(model, test_data_loader03, datasets03, test_data_loader00, datasets00):
    label_list1 = datasets03["train"].features["pos_tags"].feature.names
    label_list2 = datasets00["train"].features["pos_tags"].feature.names
    id2label1 = {str(i): label for i, label in enumerate(label_list1)}
    label2id1 = {v: k for k, v in id2label1.items()}
    id2label2 = {str(i): label for i, label in enumerate(label_list2)}
    label2id2 = {v: k for k, v in id2label2.items()}
    true_labels1 = []
    pred_labels1 = []
```



```
    true_labels2 = []
    pred_labels2 = []
    model.eval()
    with torch.no_grad():
        for idx, data in enumerate(test_data_loader03):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            pos = data['pos'].type(torch.IntTensor).to(device)

            outputs1, outputs2, *_ = model(ids, mask)
            pred1 = torch.argmax(torch.softmax(outputs1, dim=2), dim=-1).to(device, dtype=torch.int8)

            true_labels1.append(pos.cpu().detach().numpy().tolist())
            pred_labels1.append(pred1.cpu().detach().numpy().tolist())
        for idx, data in enumerate(test_data_loader00):
            ids = data['input_ids'].to(device, dtype=torch.long)
            mask = data['attention_mask'].to(device, dtype=torch.long)
            pos = data['pos'].type(torch.IntTensor).to(device)

            outputs1, outputs2, *_ = model(ids, mask)
            pred2 = torch.argmax(torch.softmax(outputs2, dim=2), dim=-1).to(device, dtype=torch.int8)

            true_labels2.append(pos.cpu().detach().numpy().tolist())
            pred_labels2.append(pred2.cpu().detach().numpy().tolist())
    t1 = result_to_list(true_labels1)
    t2 = result_to_list(true_labels2)
    p1 = result_to_list(pred_labels1)
    p2 = result_to_list(pred_labels2)
    print(compute_metrics(p1, t1, label_list1))
    print(compute_metrics(p2, t2, label_list2))
```

A.6: testers.py