Transfer Learning of BERT: Using DNABERT for the Classification of Enhancer Sequences

Sawyer Lehman, Nicholas Dibley

May 7, 2023

1 Introduction

Enhancer regions are DNA sequences that can activate or enhance the transcription of nearby genes. These regions can be located upstream, downstream, or within the introns of the gene they regulate, and they can be far away from the transcription start site. Enhancer regions contain specific DNA sequences that can bind to transcription factors and other regulatory proteins, which in turn can influence the activity of the nearby genes. These regions are often identified by profiling specific epigenetic marks and using chromatin immunoprecipitation coupled with Next-generation sequencing (NGS), specifically ChIP-Seq [1]. While ChIP-Seq can be used to identify enhancer regions, there are several downsides to using this technique for this purpose. ChIP-Seq has a limited resolution, meaning that it may not be able to precisely identify the boundaries of enhancer regions. Enhancers can be several kilobases long, and ChIP-Seq may only be able to identify the location of the transcription factor binding site within that region. ChIP-Seq can also produce false positives, where it identifies regions that are not actually enhancers but are bound by the transcription factor due to nonspecific interactions or experimental artifacts. In addition, enhancer regions are often tissue-specific, meaning that they are active only in certain cell types. ChIP-Seq experiments can be biased towards specific cell types, leading to the identification of enhancers that are not relevant in other cell types. Lastly, functional validation is often required to confirm that these regions actually act as enhancers in vivo [2]. This can be a challenging and time-consuming process.

Unlike in some other regions of the genome, enhancer sequences do not contain well-defined consensus motifs. Instead, enhancers can contain a diverse range of motifs, which makes it challenging to identify the most relevant ones based on frequency alone. Additionally, as mentioned before, the activity of enhancers is context-specific and can vary across cell types and developmental stages as well. Many computational tools used to identify motifs in enhancer sequences generate a large number of false positives, such as correlation based algorithms in [3]. These false positives can make it difficult to distinguish between biologically relevant motifs and noise.

1.1 Bioinformatics Research Problem

Due to these limitations in technology and challenges, the next-best alternative is to attempt to make predictions about whether or not a sequence is an enhancer. Certain pre-trained deep learning language models developed for Natural Language Processing (NLP) are capable of zero-shot learning and with fine-tuning, these models can perform tasks that they were not specifically trained for [4]. Furthermore, it has been shown that these language models are able to complete tasks with DNA sequences, such as non-coding regions [5] [6]. The identification and prediction of non-coding regulatory DNA sequences, such as enhancers, that are located many base pairs away from the target gene with a deep learning model would be beneficial. This leads to the question: *Can fine-tuned pre-trained deep learning language models identify enhancers?* We aim to examine the transfer learning abilities of language models intended for non-coding DNA sequences with the goal of classifying enhancer sequences.

2 Machine Learning Methods

The language model that we aim to use for the task of classification of enhancer sequences is the DNABERT model [6]. This model is a variation of BERT [7] that has been pre-trained on non-coding regions of DNA sequences. This model, like many other BERT models, is originally trained for the task of Masked Language Modeling (MLM), which requires the model to predict a masked word, or token, based on the context of all tokens in the sentence surrounding the masked token in both directions. The process for MLM is first, the sequence is tokenized and input into the BERT model, which has many layers of encoders. One of the token inputs in the sequence is masked, and the model predicts the highest probable token in the masked position based on the predefined vocabulary available.

This model can be used for tasks other than MLM with proper fine-tuning. Fine-tuning a pre-trained model allows the original weights to influence the re-training, and the model is slightly updated so that it can perform the desired task. More specifically, the original pre-trained head of the model is discarded, but the body of the model is kept. The weights of the head are then randomly initialized to the specific task that you intend to fine-tune the model for. Finally, the weights of this new head are trained for the specific task with the training data to transfer the knowledge of the pre-trained model to the task.

3 Data

The original source of the dataset we used is curated from the Ensembl database. The data is from [8], which is a repository of genomic data benchmarks such as promoters, enhancers, and open chromatin regions from three organisms: humans, mice, and fruit flies. For the purposes of this report, we are interested in human enhancer regions. The enhancer sequence data is labeled,

and the negative samples are randomly sampled sequences from the organism that are non-overlapping with the enhancer sequences. In total, there are a little over 130,000 sequences. About 100,000 were used for training, with balanced classes: 50,000 sequences of positive samples and 50,000 negative samples. The remaining 30,000 was used for testing, which also has balanced classes: 15,000 positive samples and 15,000 negative samples. The sequences were "tokenized" into 6-mers and encoded so that they could be input into the model.

Some preliminary cleaning was done to the data, shown in Listing 2 in 8. This was mainly removing any sequences in the training data that were also in the testing data to ensure unbiased evaluations of the models explained in 4.1 and 5. Additionally, a smaller subset, n=1000, of the training data with 500 positive and 500 negative samples was made to do a small hyperparameter search explained in 4.1.

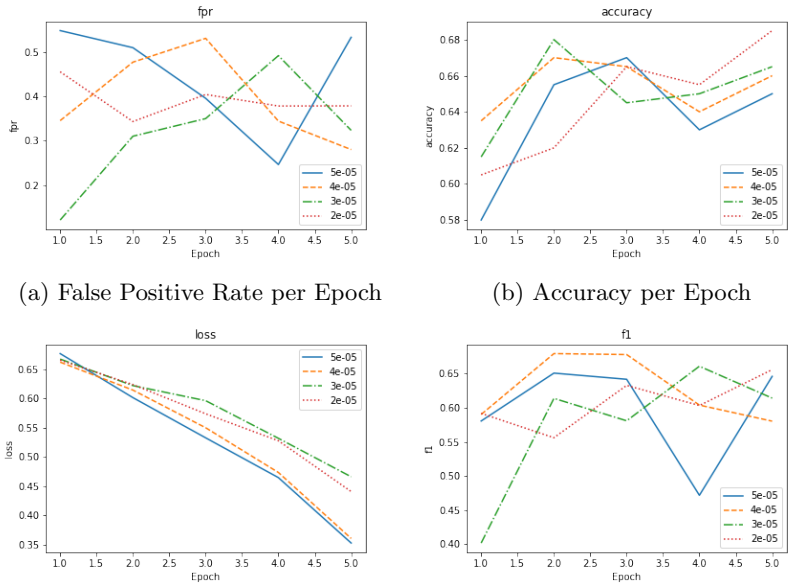
4 Experimental Design

Our overall goal in the project was to test the transferability of DNABERT to a different task. To do this, we first tested the model before fine-tuning the classifier to serve as a baseline performance. Then, a small hyperparameter search was done to see what the best strategy would be to fine-tune the model. Finally, the fine-tuned model was tested and compared to the performance of the original model.

4.1 Hyperparameter search

The first step in fine-tuning the DNABERT model is to choose certain hyperparameters, code shown in Listing 4 in Section 8. To do this, a subset was taken from the training dataset, explained in Section 3. From this subset, 80 percent was used for training and the other 20 percent was used for validation. The learning rates that were considered were $5e-5$, $4e-5$, $3e-5$, and $2e-5$, which are the same Adam optimizer learning rates found in [7]. Each round of fine-tuning with the different learning rates was done for 5 epochs and the performances on the validation set for each epoch were recorded. Another parameter that affects the model training is the batch size, but due to limitations, we could only use a batch size of 8.

Based on this small hyperparameter search, we can see that the model trained with a learning rate of $3e-05$ at 2 epochs had the best accuracy (Figure 1b) and false positive rate (Figure 1a). Although this model with these hyperparameters did not perform the best in terms of the training loss (Figure 1c) and F1 score (Figure 1d), we know that oftentimes the loss on a validation set is somewhat inflated. In addition, it is sometimes beneficial to have a lower F1 score during validation since if this were too high, it could indicate that the model is overfitting.



(a) False Positive Rate per Epoch

(b) Accuracy per Epoch

(c) Training Loss per Epoch

(d) F1 Score per Epoch

Figure 1: Validation metrics for DNABERT model fine-tuned at learning rates of $5e-5$ (blue solid line), $4e-5$ (orange dashed), $3e-5$ (green dot-dashed), and $2e-5$ (red dotted) over 5 epochs.

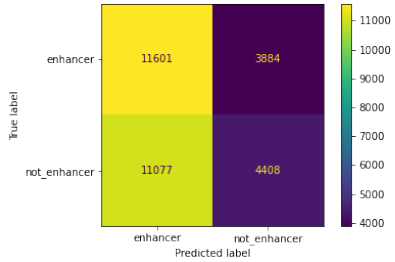
5 Experimental Results

The model was first evaluated before fine-tuning with the testing set, to see the baseline performance. The performances of both models are shown in Table 1. The original model with no updated weights to the classifier was neither accurate nor precise, with an accuracy of 51.7% and a precision of 51.1%. The recall, which is also known as the true positive rate or sensitivity, is relatively high at 74.9%. Upon examining the confusion matrix, the model is predicting a lot of false positives, which contributes to the sensitivity of the model (Figure 2a).

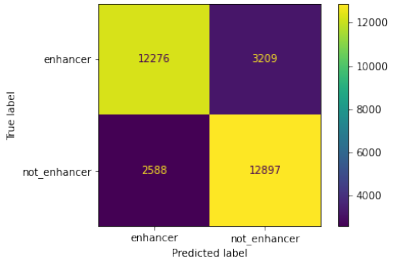
After fine-tuning the model with the full training dataset, with a learning rate of $3e-5$ at 2 epochs, the same testing set was used to evaluate the performance. The accuracy and precision were improved greatly to 81.3% and 82.6% respectively. The recall and specificity also improved to 79.3% and 83.3% respectively. Examining the confusion matrix revealed that the model is able to predict true negatives more than true positives, which contributes to the specificity of this model (Figure 2b).

metric	original	fine-tuned
Accuracy	0.517	0.813
Precision	0.511	0.826
Recall	0.749	0.793
Specificity	0.285	0.833

Table 1: Evaluation results of DNABERT model before (original) and after (fine-tuned) fine-tuning.



(a) Confusion matrix of predictions **before** fine-tuning.



(b) Confusion matrix of predictions **after** fine-tuning.

Figure 2: Confusion matrices of predictions.

6 Conclusion

Based on the results in Section 5, we see that the fine-tuned version of the DNABERT model is capable of transferring the pre-trained knowledge to the task well. We can tell that the fine-tuning made a difference because the performance on the testing dataset before the model head was fine-tuned was essentially random guessing, which is to be expected since the weights of the classification head added to the model were initialized with randomized weights as mentioned in Section 2. The model before fine-tuning was much more sensitive, which follows the character of producing a lot of false positives much like the other models for enhancer classification. After fine-tuning, the model was able to predict more true negatives, making it a more specific model.

7 Discussion

Overall, the benefit of using a pre-trained model such as BERT is the ability to transfer across tasks or even domains. This is particularly useful if the dataset is too small or noisy for a fully-fledged deep-learning model on its own. This characteristic is also beneficial because it mitigates the need for feature engineering and domain-specific knowledge that is required when creating an entirely new deep-learning model. A challenge that could arise with fine-tuning is if the data is too different from the original pre-training data, which would lead to a poorly underfitting model. Another challenge when fine-tuning, or training a deep learning model in general, is that it is computationally expensive. While attempting to fine-tune the model, we suffered greatly from the lack of GPU resources available to us, hence why we were limited to a smaller batch size. Lastly, the overall lack of interpretability of the decisions made by the model creates a problem when trying to understand what features are important in your data.

References

[1] S. Blinka, M. H. Reimer, K. Pulakanti, L. Pinello, G.-C. Yuan, and S. Rao, "Identification of transcribed enhancers by genome-wide chromatin immunoprecipitation sequencing," *Enhancer RNAs: Methods and Protocols*, pp. 91–109, 2017.

[2] A. Visel, S. Minovitsky, I. Dubchak, and L. A. Pennacchio, "Vista enhancer browser—a database of tissue-specific human enhancers," *Nucleic acids research*, vol. 35, no. suppl_1, pp. D88–D92, 2007.

[3] J. M. Hariprakash and F. Ferrari, "Computational biology solutions to identify enhancers-target gene pairs," *Computational and structural biotechnology journal*, vol. 17, pp. 821–831, 2019.

[4] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," *arXiv preprint arXiv:2109.01652*, 2021.

[5] G. Benegas, S. S. Batra, and Y. S. Song, "Dna language models are powerful zero-shot predictors of non-coding variant effects," *bioRxiv*, pp. 2022–08, 2022.

[6] Y. Ji, Z. Zhou, H. Liu, and R. V. Davuluri, "Dnabert: pre-trained bidirectional encoder representations from transformers model for dna-language in genome," *Bioinformatics*, vol. 37, no. 15, pp. 2112–2120, 2021.

[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] K. Gresova, V. Martinek, D. Cechak, P. Simecek, and P. Alexiou, "Genomic benchmarks: A collection of datasets for genomic sequence classification," *bioRxiv*, pp. 2022–06, 2022.

8 Appendix

```
# Imports:
import pandas as pd
import numpy as np
import random
from tqdm import trange
from tabulate import tabulate
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn import metrics
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from datasets import load_dataset
from transformers import AutoTokenizer,
    AutoModelForSequenceClassification, AdamW
from torch.utils.data import TensorDataset, DataLoader,
    RandomSampler, SequentialSampler

# Functions:
def data_to_df(data, subset):
    """huggingface raw data to df,
    subset can be 'train' or 'test'
    """
    lst_data = []
    for i in range(data[subset].num_rows):
        seq = data[subset][i]['seq']
        label = data[subset][i]['label']
        lst_data.append([seq, label])

    df = pd.DataFrame(lst_data, columns=['seq', 'label'])
    return df

def preprocessing(input_text, tokenizer):
    '''
  Returns <class transformers.tokenization_utils_base.BatchEncoding
    > with the following fields:
    - input_ids: list of token ids
    - token_type_ids: list of token type ids
    - attention_mask: list of indices (0,1) specifying which tokens
     should considered by the model (return_attention_mask = True).
    '''
    return tokenizer.encode_plus(
                        input_text,
                        add_special_tokens = True,
                        max_length = 128,
                        padding='max_length',
                        return_attention_mask = True,
                        truncation=True,
                        return_tensors = 'pt'
                        )

def print_rand_sentence_encoding(text):
    '''
    Displays tokens, token IDs and attention mask of a random text
    sample
```

```
    '''
    index = random.randint(0, len(text) - 1)
    tokens = tokenizer.tokenize(tokenizer.decode(token_id[index]))
    token_ids = [i.numpy() for i in token_id[index]]
    attention = [i.numpy() for i in attention_masks[index]]

    table = np.array([tokens, token_ids, attention]).T
    print(tabulate(table,
                   headers = ['Tokens', 'Token IDs', 'Attention
    Mask'],
                   tablefmt = 'fancy_grid')
         )


def b_tp(preds, labels):
    '''
    Returns True Positives (TP): count of correct predictions of
    actual class 1
    '''
    return sum([preds == labels and preds == 1 for preds, labels in
     zip(preds, labels)])

def b_fp(preds, labels):
    '''
    Returns False Positives (FP): count of wrong predictions of
    actual class 1
    '''
    return sum([preds != labels and preds == 1 for preds, labels in
     zip(preds, labels)])

def b_tn(preds, labels):
    '''
    Returns True Negatives (TN): count of correct predictions of
    actual class 0
    '''
    return sum([preds == labels and preds == 0 for preds, labels in
     zip(preds, labels)])

def b_fn(preds, labels):
    '''
    Returns False Negatives (FN): count of wrong predictions of
    actual class 0
    '''
    return sum([preds != labels and preds == 0 for preds, labels in
     zip(preds, labels)])

def b_metrics(preds, labels):
    '''
  Returns the following metrics:
    - accuracy    = (TP + TN) / N
    - precision   = TP / (TP + FP)
    - recall      = TP / (TP + FN)
    - false positive rate = FP / (FP + TN)
    - f1 = (2 * TP) / ((2 * TP) + FP + FN)
    '''
    preds = np.argmax(preds, axis = 1).flatten()
    labels = labels.flatten()
```

```
    tp = b_tp(preds, labels)
    tn = b_tn(preds, labels)
    fp = b_fp(preds, labels)
    fn = b_fn(preds, labels)

    b_accuracy = (tp + tn) / (tp + tn + fp + fn)
    b_precision = tp / (tp + fp) if (tp + fp) > 0 else 'nan'
    b_recall = tp / (tp + fn) if (tp + fn) > 0 else 'nan'
    #b_specificity = tn / (tn + fp) if (tn + fp) > 0 else 'nan'
    b_fpr = fp / (fp + tn) if (fp + tn) > 0 else 'nan'
    b_f1 = (2*tp) / ((2*tp) + fp + fn) if ((2*tp) + fp + fn) > 0
    else 'nan'

    return b_accuracy, b_precision, b_recall, b_fpr, b_f1

def train_dna_bert(model, train_dataloader, validation_dataloader,
    epochs):
    # Recommended number of epochs: 2, 3, 4. See: https://arxiv.org
    /pdf/1810.04805.pdf
    epochs = epochs

    performance = []

    for _ in trange(epochs, desc = 'Epoch'):

        # ========== Training ==========

        # Set model to training mode
        model.train()

        # Tracking variables
        tr_loss = 0
        nb_tr_examples, nb_tr_steps = 0, 0

        for step, batch in enumerate(train_dataloader):
            batch = tuple(t.to(device) for t in batch)
            b_input_ids, b_input_mask, b_labels = batch
            optimizer.zero_grad()
            # Forward pass
            train_output = model(b_input_ids,
                                 token_type_ids = None,
                                 attention_mask = b_input_mask,
                                 labels = b_labels)
            # Backward pass
            train_output.loss.backward()
            optimizer.step()
            # Update tracking variables
            tr_loss += train_output.loss.item()
            nb_tr_examples += b_input_ids.size(0)
            nb_tr_steps += 1

        # ========== Validation ==========

        model.eval()

        val_accuracy = []
        val_precision = []
```

```
        val_recall = []
        val_fpr = []
        val_f1 = []

        for batch in validation_dataloader:
            batch = tuple(t.to(device) for t in batch)
            b_input_ids, b_input_mask, b_labels = batch
            with torch.no_grad():
              # Forward pass
              eval_output = model(b_input_ids,
                                  token_type_ids = None,
                                  attention_mask = b_input_mask
                                 )

            logits = eval_output.logits.detach().cpu().numpy()
            label_ids = b_labels.to('cpu').numpy()

            b_accuracy, b_precision, b_recall, b_fpr, b_f1 =
    b_metrics(logits, label_ids)
            val_accuracy.append(b_accuracy)
            if b_precision != 'nan': val_precision.append(
    b_precision)
            if b_recall != 'nan': val_recall.append(b_recall)
            if b_fpr != 'nan': val_fpr.append(b_fpr)
            if b_f1 != 'nan': val_f1.append(b_f1)

        print('\n\t - Train loss: {:.4f}'.format(tr_loss /
    nb_tr_steps))
        print('\t - Validation Accuracy: {:.4f}'.format(sum(
    val_accuracy)/len(val_accuracy)))
        print('\t - Validation Precision: {:.4f}'.format(sum(
    val_precision)/len(val_precision)) if len(val_precision)>0 else
     '\t - Validation Precision: NaN')
        print('\t - Validation Recall: {:.4f}'.format(sum(
    val_recall)/len(val_recall)) if len(val_recall)>0 else '\t -
    Validation Recall: NaN')
        print('\t - Validation FPR: {:.4f}'.format(sum(val_fpr)/len
    (val_fpr)) if len(val_fpr)>0 else '\t - Validation FPR: NaN')
        print('\t - Validation F1: {:4f}\n'.format(sum(val_f1)/len(
    val_f1))) if len(val_f1)>0 else '\t - Validation F1: NaN'

        performance.append({'loss':tr_loss / nb_tr_steps,
                            'accuracy':sum(val_accuracy)/len(
    val_accuracy),
                            'precision':sum(val_precision)/len(
    val_precision) if len(val_precision)>0 else 0,
                            'recall':sum(val_recall)/len(val_recall
    ) if len(val_recall)>0 else 0,
                            'fpr':sum(val_fpr)/len(val_fpr) if len(
    val_fpr)>0 else 0,
                            'f1':sum(val_f1)/len(val_f1) if len(
    val_f1)>0 else 0
                            }
                           )

    return performance

```

```
def make_performance_list(x):
    loss = []
    acc = []
    prec = []
    recall = []
    fpr = []
    f1 = []

    for epoch in x:
        loss.append(epoch['loss'])
        acc.append(epoch['accuracy'])
        prec.append(epoch['precision'])
        recall.append(epoch['recall'])
        fpr.append(epoch['fpr'])
        f1.append(epoch['f1'])

    return loss, acc, prec, recall, fpr, f1

def make_plot(title, x1, x2, x3, x4, y):
    plt.plot(y, x1, label = '5e-05', linestyle='-')
    plt.plot(y, x2, label = '4e-05', linestyle='--')
    plt.plot(y, x3, label = '3e-05', linestyle='-.')
    plt.plot(y, x4, label = '2e-05', linestyle=':')
    plt.legend()
    plt.title(title)
    plt.xlabel('Epoch')
    plt.ylabel(title)
    file_name = './visualizations/param_search_'+title+'.png'
    plt.savefig(file_name)
    plt.show()

def test_metrics(df):
    pos = df[df['true_label']=='enhancer']
    tp = len(pos[pos['prediction']=='enhancer'])
    fn = len(pos[pos['prediction']=='not_enhancer'])
    neg = df[df['true_label']=='not_enhancer']
    fp = len(neg[neg['prediction']=='enhancer'])
    tn = len(neg[neg['prediction']=='not_enhancer'])

    print('accuracy: ', ((tp+tn)/(tp+tn+fp+fn)))
    print('precision: ', (tp/(tp+fp)))
    print('recall: ', (tp/(tp+fn)))
    print('specificity: ', (tn/(tn+fp)))
    print('f1 score: ', ((2*tp)/((2*tp)+fp+fn)))
```

Listing 1: Imports and functions

```
# Preparing data:
# loading data from huggingface
raw_data = load_dataset("katarinagresova/
    Genomic_Benchmarks_human_enhancers_ensembl")
# converting huggingface raw data to df
train = data_to_df(raw_data, 'train')
test = data_to_df(raw_data, 'test')
# getting overlapping samples in the train and test
match_lst = []
for x in train['seq'].values.tolist():
    for y in test['seq'].values.tolist():
```

```
        if x==y:
            idx = train.index[train['seq'] == x].tolist()
            match_lst.extend(idx)
# removing the matching samples from the training set
train = train.drop(match_lst)
# creating a training subset with 500 positive and 500 negative samples
pos_train = train[train['label'] == 1].sample(n=500, random_state=42) # getting random sample of 500 positive samples
neg_train = train[train['label'] == 0].sample(n=500, random_state=42) # getting random sample of 500 negative samples
train_subset = pos_train.append(neg_train, ignore_index=True).sample(frac=1, random_state=42) # combining and shuffling
# writing data to csv files
train.to_csv('./data/train.csv', index=False)
test.to_csv('./data/test.csv', index=False)
train_subset.to_csv('./data/train_subset.csv', index=False)
```

Listing 2: Preparing Data.

```
# setting device to use gpu
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# loading DNABERT model and putting model to gpu
id2label = {0: "not_enhancer", 1: "enhancer"}
label2id = {"not_enhancer": 0, "enhancer": 1}
tokenizer = AutoTokenizer.from_pretrained("zhihan1996/DNA_bert_6", trust_remote_code=True)
model = AutoModelForSequenceClassification.from_pretrained("zhihan1996/DNA_bert_6", num_labels=2,
                                                           id2label=id2label, label2id=label2id,

    trust_remote_code=True
                                                          )
model.cuda()
# loading test dataset
test = pd.read_csv('./data/test.csv')
pos = test[test['label']==1]
neg = test[test['label']==0]
neg = neg.sample(n=len(pos), random_state=42)
test = pd.concat([pos, neg], ignore_index=True)
test_text = test.seq.values.tolist()
# formatting sequences to 6-mers
test_text = [' '.join([seq[i:i+6] for i in range(0, len(seq), 6)]) for seq in test_text]
test_labels = test.label.values.tolist()
test_output = []
for seq, lab in zip(test_text, test_labels):
    test_ids = []
    test_attention_mask = []
    encoding = preprocessing(seq, tokenizer)

    test_ids.append(encoding['input_ids'])
    test_attention_mask.append(encoding['attention_mask'])
    test_ids = torch.cat(test_ids, dim = 0)
    test_attention_mask = torch.cat(test_attention_mask, dim = 0)

```

```
    with torch.no_grad():
        output = model(test_ids.to(device), token_type_ids = None,
    attention_mask = test_attention_mask.to(device))

    prediction = 'enhancer' if np.argmax(output.logits.cpu().numpy
    ()).flatten().item() == 1 else 'not_enhancer'

    if lab == 1:
        true_lab = 'enhancer'
    else:
        true_lab = 'not_enhancer'

    test_output.append([seq, true_lab, prediction])

results = pd.DataFrame(test_output, columns = ['sequence', '
    true_label', 'prediction'])
results.to_csv('zero_shot_predictions.csv', index=False)
```

Listing 3: Testing model before finetuning.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu
    ')
tokenizer = AutoTokenizer.from_pretrained("zhihan1996/DNA_bert_6",
    trust_remote_code=True)
# Loading subset for hyperparameter search
train = pd.read_csv('./data/train_subset.csv')
train_text = train.seq.values.tolist()
# formatting sequences to 6-mers:
train_text = [' '.join([seq[i:i+6] for i in range(0, len(seq), 6)])
     for seq in train_text]
train_labels = train.label.values.tolist()
# checking to make sure tokenization worked
print_rand_sentence_encoding(train_text)
# tokenizing train subset
token_id = []
attention_masks = []

for sample in train_text:
    encoding_dict = preprocessing(sample, tokenizer)
    token_id.append(encoding_dict['input_ids'])
    attention_masks.append(encoding_dict['attention_mask'])


token_id = torch.cat(token_id, dim = 0)
attention_masks = torch.cat(attention_masks, dim = 0)
labels = torch.tensor(train_labels)

val_ratio = 0.2
batch_size = 8

# splitting the training into train and validation
train_idx, val_idx = train_test_split(
    np.arange(len(labels)),
    test_size = val_ratio,
    shuffle = True,
    stratify = labels)

train_set = TensorDataset(token_id[train_idx],
```

```
                           attention_masks[train_idx],
                           labels[train_idx])

val_set = TensorDataset(token_id[val_idx],
                        attention_masks[val_idx],
                        labels[val_idx])

# wrapping in dataloader object
train_dataloader = DataLoader(
            train_set,
            sampler = RandomSampler(train_set),
            batch_size = batch_size
        )

validation_dataloader = DataLoader(
            val_set,
            sampler = SequentialSampler(val_set),
            batch_size = batch_size
        )

id2label = {0: "not_enhancer", 1: "enhancer"}
label2id = {"not_enhancer": 0, "enhancer": 1}

# Recommended learning rates (Adam): 5e-5, 3e-5, 2e-5. See: https
    ://arxiv.org/pdf/1810.04805.pdf
learning_rates = [5e-05, 4e-05, 3e-05, 2e-05]
performance = {}

for lr in learning_rates:
    model = AutoModelForSequenceClassification.from_pretrained("
    zhihan1996/DNA_bert_6", num_labels=2,
                                                               id2label
    =id2label, label2id=label2id,

    trust_remote_code=True
                                                              )

    #model = nn.DataParallel(model)

    optimizer = torch.optim.AdamW(model.parameters(), lr = lr, eps
    = 1e-08)

    model.cuda()

    output = train_dna_bert(model, train_dataloader,
    validation_dataloader, epochs=5)
    performance[str(lr)] = output
    torch.cuda.empty_cache()

y = [1, 2, 3, 4, 5]
x1=performance['5e-05']
x2=performance['4e-05']
x3=performance['3e-05']
x4=performance['2e-05']

x1_loss, x1_acc, x1_prec, x1_recall, x1_fpr, x1_f1 =
    make_performance_list(x1)
```

```
x2_loss, x2_acc, x2_prec, x2_recall, x2_fpr, x2_f1 =
    make_performance_list(x2)
x3_loss, x3_acc, x3_prec, x3_recall, x3_fpr, x3_f1 =
    make_performance_list(x3)
x4_loss, x4_acc, x4_prec, x4_recall, x4_fpr, x4_f1 =
    make_performance_list(x4)

make_plot('loss', x1_loss, x2_loss, x3_loss, x4_loss, y)
make_plot('accuracy', x1_acc, x2_acc, x3_acc, x4_acc, y)
make_plot('precision', x1_prec, x2_prec, x3_prec, x4_prec, y)
make_plot('recall', x1_recall, x2_recall, x3_recall, x4_recall, y)
make_plot('fpr', x1_fpr, x2_fpr, x3_fpr, x4_fpr, y)
make_plot('f1', x1_f1, x2_f1, x3_f1, x4_f1, y)
```

Listing 4: Hyperparameter search.

```
# Fine-tune
# ---------
# further data cleaning:
train = pd.read_csv('./data/train.csv')
pos = train[train['label']==1]
neg = train[train['label']==0]
neg = neg.sample(n=len(pos), random_state=42)
train = pd.concat([pos, neg], ignore_index=True)
train_text = train.seq.values.tolist()
# formatting sequences to 6-mers
train_text = [' '.join([seq[i:i+6] for i in range(0, len(seq), 6)])
     for seq in train_text]
train_labels = train.label.values.tolist()

token_id = []
attention_masks = []

for sample in train_text:
    encoding_dict = preprocessing(sample, tokenizer)
    token_id.append(encoding_dict['input_ids'])
    attention_masks.append(encoding_dict['attention_mask'])


token_id = torch.cat(token_id, dim = 0)
attention_masks = torch.cat(attention_masks, dim = 0)
labels = torch.tensor(train_labels)

val_ratio = 0.2
batch_size = 8

# splitting the training into train and validation
train_idx, val_idx = train_test_split(
    np.arange(len(labels)),
    test_size = val_ratio,
    shuffle = True,
    stratify = labels)

train_set = TensorDataset(token_id[train_idx],
                          attention_masks[train_idx],
                          labels[train_idx])

val_set = TensorDataset(token_id[val_idx],
```

```
                        attention_masks[val_idx],
                        labels[val_idx])

# wrapping in dataloader object
train_dataloader = DataLoader(
            train_set,
            sampler = RandomSampler(train_set),
            batch_size = batch_size
        )

validation_dataloader = DataLoader(
            val_set,
            sampler = SequentialSampler(val_set),
            batch_size = batch_size
        )

id2label = {0: "not_enhancer", 1: "enhancer"}
label2id = {"not_enhancer": 0, "enhancer": 1}
model = AutoModelForSequenceClassification.from_pretrained("zhihan1996/DNA_bert_6", num_labels=2,
                                                           id2label=id2label, label2id=label2id,
    trust_remote_code=True
                                                          )

optimizer = torch.optim.AdamW(model.parameters(), lr = 3e-05, eps = 1e-08)
model.cuda()
final_output = train_dna_bert(model, train_dataloader, validation_dataloader, epochs=2)
final_df = pd.DataFrame(final_output)
final_df.to_csv('./fine_tune_validation_results.csv')

# Testing
# -------
# loading test dataset
test = pd.read_csv('./data/test.csv')
pos = test[test['label']==1]
neg = test[test['label']==0]
neg = neg.sample(n=len(pos), random_state=42)
test = pd.concat([pos, neg], ignore_index=True)
test_text = test.seq.values.tolist()
# formatting sequences to 6-mers
test_text = [' '.join([seq[i:i+6] for i in range(0, len(seq), 6)]) for seq in test_text]
test_labels = test.label.values.tolist()
test_output = []
for seq, lab in zip(test_text, test_labels):
    test_ids = []
    test_attention_mask = []
    encoding = preprocessing(seq, tokenizer)

    test_ids.append(encoding['input_ids'])
    test_attention_mask.append(encoding['attention_mask'])
    test_ids = torch.cat(test_ids, dim = 0)
    test_attention_mask = torch.cat(test_attention_mask, dim = 0)
```

```

    with torch.no_grad():
        output = model(test_ids.to(device), token_type_ids = None, attention_mask = test_attention_mask.to(device))

    prediction = 'enhancer' if np.argmax(output.logits.cpu().numpy()).flatten().item() == 1 else 'not_enhancer'

    if lab == 1:
        true_lab = 'enhancer'
    else:
        true_lab = 'not_enhancer'

    test_output.append([seq, true_lab, prediction])

results = pd.DataFrame(test_output, columns = ['sequence', 'true_label', 'prediction'])
results.to_csv('./finetune_model_predictions.csv', index=False)
```

Listing 5: Fine-tuning model and testing.

```
# Zero-shot predictions
# ---------------------
zero = pd.read_csv('./zero_shot_predictions.csv', index_col=0)
test_metrics(zero)
actual = zero['true_label'].values
predicted = zero['prediction'].values
confusion_matrix = metrics.confusion_matrix(actual, predicted)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = ['enhancer', 'not_enhancer'])
cm_display.plot()
plt.savefig('./visualizations/zero_shot_cm.png')
plt.show()

# Fine-tune predictions
# ---------------------
fine = pd.read_csv('./finetune_model_predictions.csv')
test_metrics(fine)
actual = fine['true_label'].values
predicted = fine['prediction'].values
confusion_matrix = metrics.confusion_matrix(actual, predicted)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = ['enhancer', 'not_enhancer'])
cm_display.plot()
plt.savefig('./visualizations/final_test_cm.png')
plt.show()
```

Listing 6: Getting final metrics from zero-shot and fine-tune models.