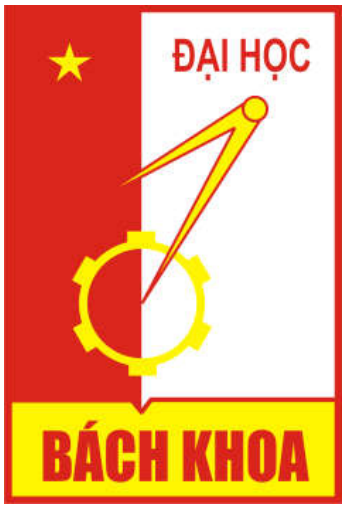
VIỆN CÔNG NGHỆ THÔNG TIN & TRUYỀN THÔNG

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI



MẠNG NEURAL VÀ ỨNG DỤNG

ĐỀ TÀI:

Predicting Protein-DNA Binding Residues

Giảng viên hướng dẫn: TS. Nguyễn Hồng Quang

Sinh viên thực hiện: Ngô Huy Hoàng 20155637
Trần Hải Đăng 20155357

Hà Nội 01/2019

LỜI GIỚI THIỆU

Mạng Neural là phương pháp học máy phổ biến và hiệu quả song song với sự phát triển của phần cứng hệ thống và các công cụ tính toán đồng thời, đặc biệt là GPU. Mạng Neural đã thể hiện được sự hiệu quả rõ rệt và kết quả vô cùng khả quan trong vô vàn các bài toán khác nhau trong đời sống, từ nhận dạng ảnh, phân biệt đối tượng, xử lý ngôn ngữ tự nhiên, robotics,... Trong những ứng dụng đó, không thể không nhắc tới đóng góp to lớn của mạng neural cho ngành sinh học. Phương pháp này đã cải thiện đáng kể các vấn đề về dự đoán bệnh, phát hiện thuốc mới, … Vì thế, trong danh sách các bài tập lớn của môn học, chúng em đã lựa chọn bài toán "Dự đoán dư lượng liên kết protein-DNA", trong quá trình giải quyết bài toán này chúng em đã tìm hiểu được rất nhiều kiến thức bổ ích về mạng neural, bao gồm các kiến thức căn bản, nền tảng của học máy như các phương pháp tối ưu hàm loss, thuật toán perceptron cơ bản, thiết kế mạng neural cơ bản, cho đến các kiến thức phức tạp hơn, ứng dụng cao hơn vào thực tế như các quy tắc thiết lập một dự án học máy, phân chia dữ liệu, điều chỉnh các siêu tham số, regularization để tránh overfiting và các kiến thức về mạng Convolution Neural Network (CNN), các đặc trưng thường gặp trong các bài toán học máy thuộc lĩnh vực sinh học và cách áp dụng các thư viện, công cụ như biopython, numpy, scikit-learn, tensorflow,...

Trong quá trình tìm hiểu và tiến hành triển khai bài tập lớn, chúng em còn gặp phải nhiều thiếu sót, mong nhận được phản hồi và góp ý từ thầy để bài tập lớn của nhóm em được đầy đủ và hoàn thiện hơn.

Chúng em xin cảm ơn thầy đã tận tình chỉ bảo và hướng dẫn!

MỤC LỤC

DANH MỤC HÌNH VẼ

DANH MỤC BẢNG BIỂU

1. MÔ TẢ BÀI TOÁN

a. Bài toán và dữ liệu

- Tương tác giữa protein và dna là không thể thiếu đối với các hoạt động sinh học và đóng vai trò qan trọng trong nhiều quá trình sinh học như sao chép DNA, phiên mã, phục hồi, …Vì thế, việc xác định chính xác dư lượng liên kết Protein-DNA đóng một vai trò quan trọng đối với việc phân tích chức năng của protein và tìm kiếm, sản xuất các loại thuốc mới.
- Đề tài này nhằm cung cấp một phương pháp hiện đại để xác định, dự đoán chính xác dư lượng liên kết Protein-DNA

b. Tiêu chí đánh giá

Số liệu để đánh giá là Accuracy (%) trên tập Public Test:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

c. Phương pháp state-of-the-art

Phương pháp được sử dụng trong paper đó là kết nối một cách có trọng số 2 feature phổ biến và hiệu quả của các phần tử trong chuỗi protein đó là PSSM và PSA và sau đó sử dụng Boosting Multiple SVMs để phân loại. Phương pháp này được gọi là TargetDNA

Kết quả của phương pháp này và đối chiếu với các phương pháp trước đó để dự đoán dư lượng liên kết Protein-DNA được thể hiện trong Bảng 1.

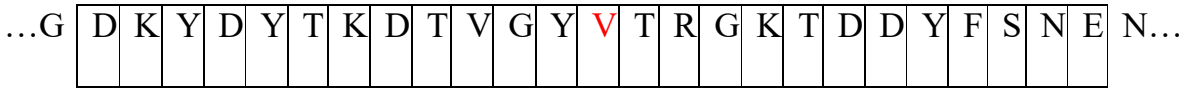
No.	Predictor	Acc
1	BindN	79.15%
2	ProteDNA	95.11%
3	BindN+ (FPR ≈ 5%)	91.58%
4	BindN+ (Spe ≈ 85%)	83.69%
5	MetaDBSite	90.41%
6	DP-Bind	81.40%

7	DNABind	79.78%
8	**TargetDNA (Spe ≈ Sen)**	**84.52%**
9	**TargetDNA (FPR ≈5%)**	**90.89%**

Bảng 1. So sánh kết quả giữa TargetDNA và các phương pháp dự đoán khác cho dư lượng liên kết protein-DNA trên tập test PDNA-TEST

d. Phương pháp đề xuất

- Trong phạm vi bài tập lớn lần này, chúng em không follow theo paper vì thuật toán được sử dụng trong paper là SVM (cụ thể là Boosting Multiple SVMs), chúng em sẽ triển khai sử dụng convolution neraul network để giải quyết bài toán này.
- Dựa trên sự tham khảo của các phương pháp được đề xuất trên một số paper khác, cộng thêm việc các công cụ để trích xuất PSSM feature và PSA feature trong paper gốc đều không còn hoạt động được nên nhóm quyết định trích xuất feature theo cách khác. Trong bài toán này chúng em sẽ trích xuất feature của data bằng việc sử dụng one-hot encoding để mã hóa mỗi residue trong chuỗi protein thành một vector 20 chiều (mỗi chiều đại diện cho một loại amino acid tiêu chuẩn).
- Vì tính chất của từng phần tử này phụ thuộc vào cả những phần tử bên cạnh nó trong chuỗi protein nên chúng em sẽ sử dụng kỹ thuật sliding window với window size bằng 25, trong đó, phần tử thứ 13 sẽ là phần tử mục tiêu được xét. (Hình 1)



Hình 1. Sliding window gồm 25 phần tử, phẩn tử trung tâm được xét là phần tử thứ 13

- Để có thể trượt từ đầu chuỗi protein đến cuối chuỗi, ta sẽ cần có cả thêm những phần tử thuộc window nhưng không nằm trong chuỗi protein đang xét (gọi là missing residue). Vì thế ta cần phải biểu diễn cả những phần tử này nữa, để có thể làm được điều này, chúng em thêm một chiều nữa cho

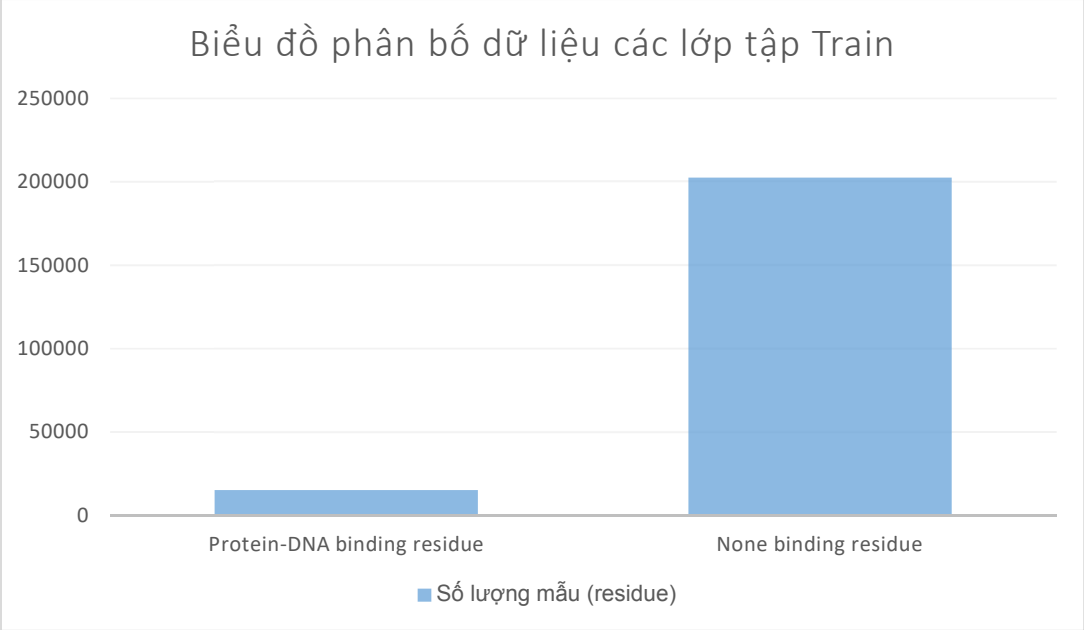
phần tử đã được one-hot encoded là chiều thứ 21, chiều này thể hiện phần tử đang xét có nằm trong chuỗi protein hay không. Vậy tổng cộng chiều của mỗi feature được tạo ra là: 25x21

- Từ đó, những feature đã được trích xuất sẽ được feed vào một mạng CNN để training

2. Dataset

a. Training set và testing set

- Format của dataset là dạng FASTA. Định dạng FASTA là một định dạng dựa trên văn bản để biểu diễn trình tự nucleotide hoặc chuỗi peptid, trong đó nucleotide hoặc axit amin được biểu diễn bằng cách sử dụng các mã đơn (ARNDCQEGHILKMFPSTWYV).
- Để xử lý và cấu trúc data cho loại định dạng này, chúng em sẽ sử dụng package Biopython
- Data set được xây dựng dựa trên bộ dữ liệu của 7168 chuỗi protein liên kết DNA trong Protein data bank (PDB). Sau khi loại bỏ các chuỗi thừa thãi bằng phần mềm CD-hit thì thu được tổng cộng 584 chuỗi protein. Sau đó chuỗi này được chia thành 2 phần:
    - Tập dữ liệu cho training là tập PDNA-543 gồm 543 chuỗi protein, trong đó có 9549 dư lượng liên kết DNA (dương tính) và 134995 dư lượng không liên kết (âm tính), tuy nhiên, nhóm em đã kết hợp thêm một bộ dữ liệu trong một paper khác (được thầy cung cấp) là tập metaDBsite-PDNA-316 để tạo thành một tập data mới gồm 859 chuỗi protein, trong đó có 202104 mẫu âm tính và 15158 mẫu dương tính theo như biểu diễn trên Hình 2 và Bảng 2:

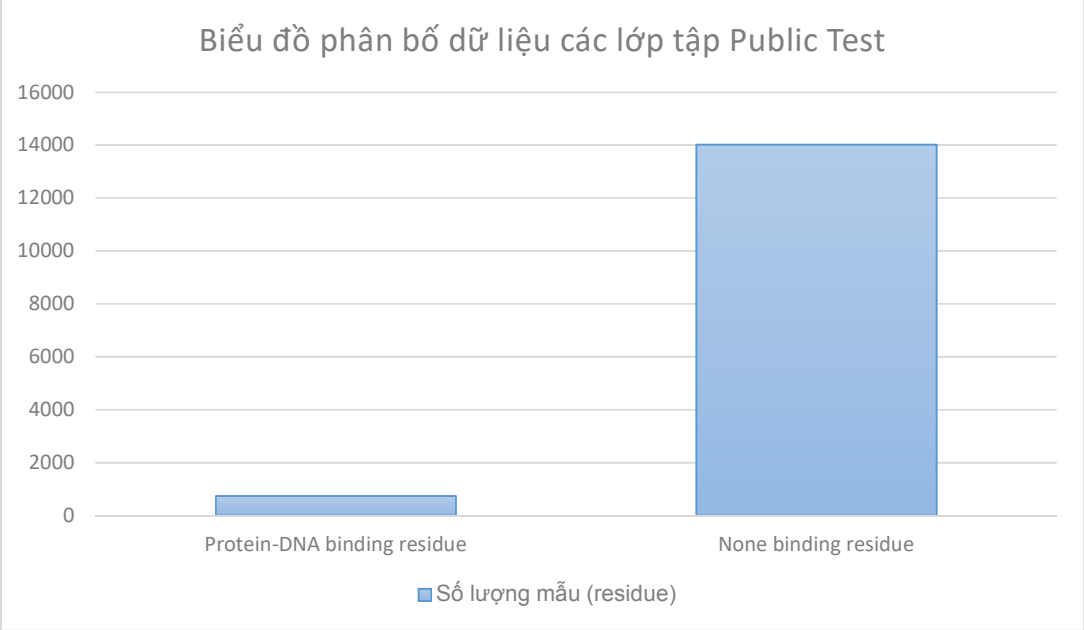


Hình 2. Biểu đồ phân bố dữ liệu giữa Protein-DNA binding residue và None binding residue trong tập train

Label	#Samples
Protein-DNA binding residue	15158
None binding residue	202104
Total	**217262**

Bảng 2: Số lượng tệp mẫu được cung cấp để dùng làm tập Train/Validation cho mô hình

- Tập dữ liệu cho testing gồm 41 chuỗi protein có 743 mẫu dương tính và 14021 mẫu âm tính được biểu diễn trong Hình 3 và Bảng 3:



Hình 3. Biểu đồ phân bố dữ liệu giữa Protein-DNA binding residue và None binding residue trong tập dữ liệu test

Label	#Samples
Protein-DNA binding residue	734
None binding residue	14021
Total	**14755**

Bảng 3: Số lượng tệp mẫu được cung cấp để dùng làm tập dữ liệu Test cho mô hình

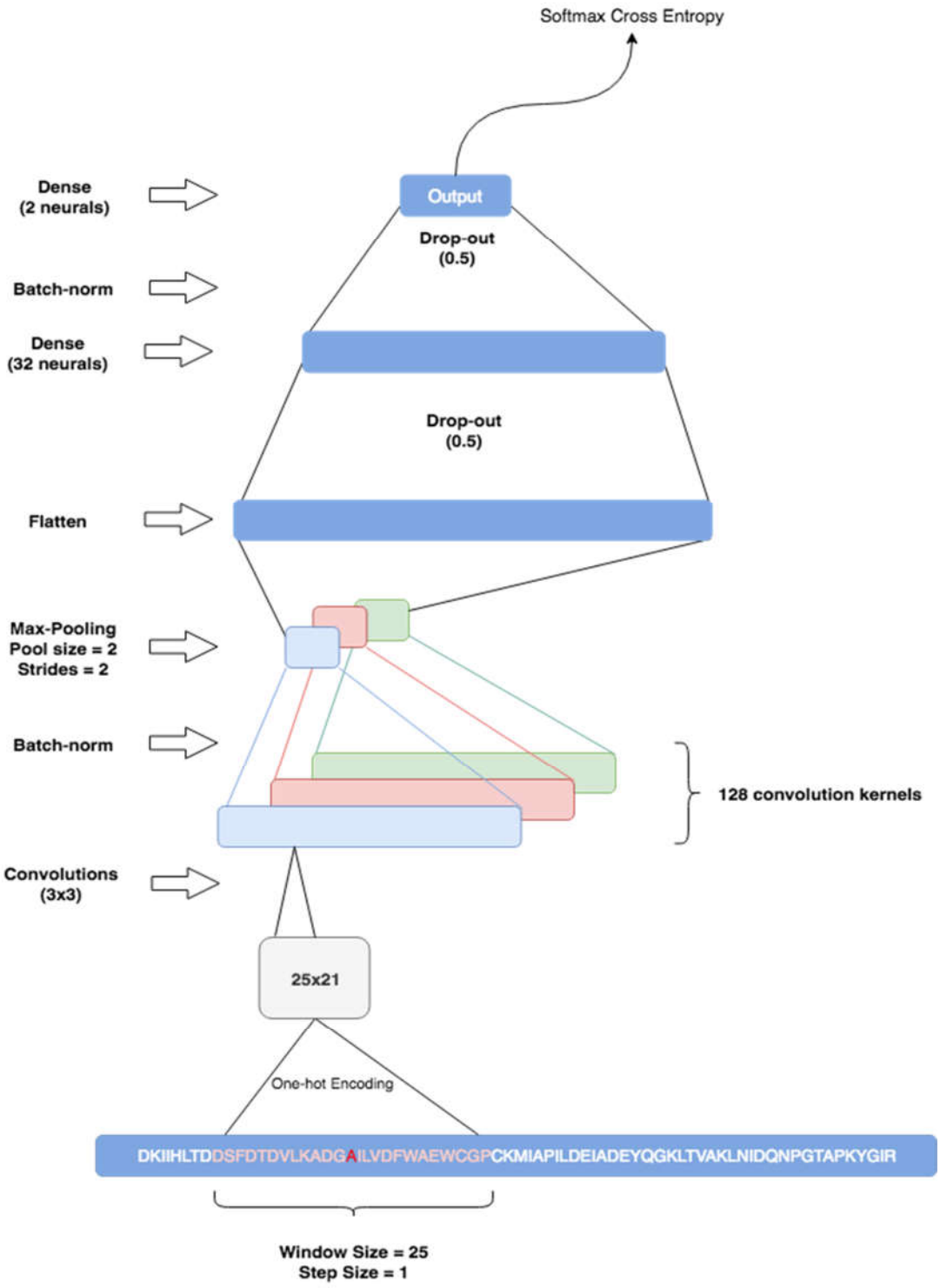
- Trong bài tập lớn này, nhóm sẽ sử dụng 5-Fold Cross Validation (4 Fold cho training, 1 Fold cho validation), train lần lượt 10 lần (10 random seeds) với mỗi Fold

b. Trích xuất đặc trưng

Phương pháp và ý tưởng trích xuất đặc trưng cho dạng dữ liệu này đã được trình bày trong phần 1.d. Phương pháp đề xuất

3. Sơ đồ kiến trúc hệ thống

- Sơ đồ kiến trúc hệ thống được biểu diễn trong Hình 4:



Hình 4. Sơ đồ kiến trúc hệ thống biểu diễn các bước xử lý và các thông số cơ bản của mô hình.

- Trong mô hình trên, các layer convolution và layer dense đầu tiên được áp dụng hàm activation là hàm ReLU
- Số các parameters:
  - Convolution layer: (3x3+1)x128 = 1,280
  - Batch-norm_1: (25x21x128+1)x2 = 134,402
  - Max pooling_1: 0
  - Flatten: 0
  - Drop-out: 0
  - Dense_1: (25x21x128+1)x32 = 2,150,432
  - Batch-norm: (32+1)x2 = 66
  - Drop-out: 0
  - Dense_2: (32+1)x2 = 66

  Tổng hợp: 2,286,246 parameters

- Với các mô hình trên, nhóm áp dụng các hyperparameter như sau:

  + **Learning rate**: 0.00001

  + **Decay steps**: 5000

  + **Batch size**: 128

  + **Number of epochs**: 20

4. KẾT QUẢ THỬ NGHIỆM

- Module preprocess gồm các hàm phục vụ cho việc tiền xử lý dữ liệu, trích xuất feature dựa trên file đầu vào (fasta)

```
# preprocess.py

# import numpy for caculating
import numpy as np
# import Bio for fasta file handling
from Bio import SeqIO
# import tensorflow
import tensorflow as tf
# import KFold for cross-validation handling
from sklearn.model_selection import KFold

# 5-fold cross-validation
KF = KFold(n_splits=5)
# window size for sliding window technique
WINDOW_SIZE = 25
# 20 standard amino acids presentation and '@' digit present the missing residue
STANDARD_AMINO_ACID = 'ARNDCQEGHILKMFPSTWYV@'
# convert standard amino acids char list to int list
char_to_int = dict((c, i) for i, c in enumerate(STANDARD_AMINO_ACID))
# convert standard amino acids int list to char list
int_to_char = dict((i, c) for i, c in enumerate(STANDARD_AMINO_ACID))



# Residue Feature Extractor
class ResidueFeatureExtractor:
    def __init__(self, name, sequence):

        # append missing residue element to sequence
        for _ in range(int(WINDOW_SIZE / 2)):
            sequence = "@" + sequence + "@"
```

```
        # integer encode the sequence
        integer_encoded_seq = self.get_integer_values_of_sequence(sequence)

        # one hot the sequence
        onehot_encoded_seq = self.get_one_hot_sequence(integer_encoded_seq)

        # feature sequence
        residue_feature_sequence = self.get_residue_feture_sequence(onehot_encoded_seq)

        # add the attributes to self
        self.name = name
        self.sequence = sequence
        self.integer = integer_encoded_seq
        self.onehot = onehot_encoded_seq
        self.features = residue_feature_sequence

    # get integer values from a sequence
    @staticmethod
    def get_integer_values_of_sequence(sequence):
        integer_encoded = [char_to_int[char] for char in sequence]
        return integer_encoded

    # one-hot encoding an integer sequence
    @staticmethod
    def get_one_hot_sequence(integer_sequence):
        # init an empty list
        one_hot_encoded_sequence = list()
        # loop through integer sequence and flag the index of each amino acid as 1 for each
amino acid
        for value in integer_sequence:
            one_hot_encoded_char = [0 for _ in range(len(STANDARD_AMINO_ACID))]
            one_hot_encoded_char[value] = 1
            one_hot_encoded_sequence.append(one_hot_encoded_char)
```

```
        return one_hot_encoded_sequence

    # convert one-hot encoded sequence to final feature sequence base on sliding-window technique
    @staticmethod
    def get_residue_feture_sequence(one_hot_encoded_sequence):
        # init an empty list
        residue_feature_sequence = list()
        # loop through the sequence
        for i in range(len(one_hot_encoded_sequence) - WINDOW_SIZE + 1):
            # each element is converted to [WINDOW_SIZE] elements around it
            residue_feature = one_hot_encoded_sequence[i: i + WINDOW_SIZE]
            # append to the final sequence
            residue_feature_sequence.append(residue_feature)
        return residue_feature_sequence


# get input features from input file path, follow the mode (training or evaluate) and cross-validation index
def get_input_features(input_file, mode, cross_val_index):
    # init an empty list
    input_features = list()
    # read the fasta sequences from input file
    fasta_sequences = SeqIO.parse(open(input_file), 'fasta')

    # loop through fasta sequences
    for fasta in fasta_sequences:
        # get name and value of each sequence
        name, sequence = fasta.id, str(fasta.seq)
        # get the ResidueFeatureExtractor object of current sequence
        extractor = ResidueFeatureExtractor(name, sequence)
        # append the feature to the list
        input_features += extractor.features
```

```
    # convert to array with data type uint8 (0-255)
    input_features = np.array(input_features, dtype=np.uint8)
    # split train-test set following k-fold cross validation
    splited_features = list(KF.split(input_features))
    # get the training set index
    train_index = splited_features[cross_val_index][0]
    # get the evaluating set index
    eval_index = splited_features[cross_val_index][1]

    # return training set in train mode
    if mode == tf.estimator.ModeKeys.TRAIN:
        return input_features[train_index]
    # return evaluating set in evalutae mode
    else:
        return input_features[eval_index]



# get input labels from input file path, follow the mode (training or evaluate) and cross-validation index
def get_input_labels(input_file, mode, cross_val_index):
    # init an empty list
    input_labels = list()
    # read the fasta sequences from input file
    fasta_sequences = SeqIO.parse(open(input_file), 'fasta')
    # loop through fasta sequences
    for fasta in fasta_sequences:
        # get the value of each sequence
        sequence = str(fasta.seq)
        # append to the list
        input_labels += list(sequence)

    # convert to array with data type uint8 (0-255)
    input_labels = np.array(input_labels, dtype=np.uint8)
    # split train-test set following k-fold cross validation
```

```
    splited_labels = list(KF.split(input_labels))
    # get the training set index
    train_index = splited_labels[cross_val_index][0]
    # get the evaluating set index
    eval_index = splited_labels[cross_val_index][1]
    # return training set in train mode
    if mode == tf.estimator.ModeKeys.TRAIN:
        return input_labels[train_index]
    # return evaluating set in train mode
    else:
        return input_labels[eval_index]



# get the test features from input file path
def get_test_features(input_file):
    # init the empty list
    input_feature = list()
    # read the fasta sequences from input file
    fasta_sequences = SeqIO.parse(open(input_file), 'fasta')

    # loop through fasta sequences
    for fasta in fasta_sequences:
        # get name and value of each sequence
        name, sequence = fasta.id, str(fasta.seq)
        # get the ResidueFeatureExtractor object of current sequence
        extractor = ResidueFeatureExtractor(name, sequence)
        # append the feature to the list
        input_feature += extractor.features
    # convert to array
    return np.array(input_feature, dtype=np.uint8)

# get the test labels from input file path
def get_test_labels(input_file):
    # init the empty list
```

```
    input_labels = list()
    # read the fasta sequences from input file
    fasta_sequences = SeqIO.parse(open(input_file), 'fasta')
    # loop through fasta sequences
    for fasta in fasta_sequences:
        # get value of each sequence
        sequence = str(fasta.seq)
        # append to the list
        input_labels += list(sequence)
    # convert to array
    return np.array(input_labels, dtype=np.uint8)        extractor =
ResidueFeatureExtractor(name, sequence)
        input_feature += extractor.features
    if mode == tf.estimator.ModeKeys.TRAIN:
        return np.array(input_feature[0:int(SPLIT_RATIO * len(input_feature))],
dtype=np.uint8)
    else:
        return np.array(input_feature[int(SPLIT_RATIO * len(input_feature)):],
dtype=np.uint8)


def get_input_labels(input_file, mode):
    input_labels = list()
    fasta_sequences = SeqIO.parse(open(input_file), 'fasta')
    for fasta in fasta_sequences:
        sequence = str(fasta.seq)
        input_labels += list(sequence)
    if mode == tf.estimator.ModeKeys.TRAIN:
        return np.array(input_labels[0:int(SPLIT_RATIO * len(input_labels))], dtype=np.uint8)
    else:
        return np.array(input_labels[int(SPLIT_RATIO * len(input_labels)):], dtype=np.uint8)


def get_test_features(input_file):
```

```
    input_feature = list()
    fasta_sequences = SeqIO.parse(open(input_file), 'fasta')

    for fasta in fasta_sequences:
        name, sequence = fasta.id, str(fasta.seq)
        extractor = ResidueFeatureExtractor(name, sequence)
        input_feature += extractor.features
    return np.array(input_feature, dtype=np.uint8)



def get_test_labels(input_file):
    input_labels = list()
    fasta_sequences = SeqIO.parse(open(input_file), 'fasta')
    for fasta in fasta_sequences:
        sequence = str(fasta.seq)
        input_labels += list(sequence)
    return np.array(input_labels, dtype=np.uint8)
```

- Module pdna gồm các hàm phục vụ cho việc đọc và chuẩn bị data để đưa vào mô hình traning:

```
# pdna.py

# Import dependences for python 2.7
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# import numpy for caculating
import numpy as np
# import tensorflow
import tensorflow as tf
# import preprocess module
import preprocess
```

```
# datasets directory
LOCAL_DIR = "data/TargetDNA/"
# input train sequences file name
INPUT_TRAIN_SEQUENCE_FILE = 'PDNA-859_sequence.fasta'
# input train labels file name
INPUT_TRAIN_LABEL_FILE = 'PDNA-859_label.fasta'
# input test sequences file name
INPUT_TEST_SEQUENCE_FILE = 'PDNA-TEST_sequence.fasta'
# input test labels file name
INPUT_TEST_LABEL_FILE = 'PDNA-TEST_label.fasta'

# number of classes
NUM_CLASSES = 2

# Get addition dataset params.
def get_params():
    return {
        "num_classes": NUM_CLASSES,
    }

# This function will be called once to prepare the dataset.
def prepare():
    """Do some addition things here"""

#Create an instance of the dataset object.
def read(split, is_test, cross_val_index):

    # get the sequence list and labels in testing phase
    if(is_test):
        # get sequence list
        sequence = preprocess.get_test_features(LOCAL_DIR +
INPUT_TEST_SEQUENCE_FILE)
        print("Loaded %d TEST residue fetures." % len(sequence))
```

```
        # get labels
        labels = preprocess.get_test_labels(LOCAL_DIR + INPUT_TEST_LABEL_FILE)
        print("Loaded %d TEST labels" % len(labels))
    # get the sequence list and labels in training or evaluating phase
    else:
        # get sequence list
        sequence = preprocess.get_input_features(LOCAL_DIR +
INPUT_TRAIN_SEQUENCE_FILE, split, cross_val_index)
        print("Loaded %d residue fetures." % len(sequence))

        # get labels
        labels = preprocess.get_input_labels(LOCAL_DIR + INPUT_TRAIN_LABEL_FILE,
split, cross_val_index)
        print("Loaded %d labels" % len(labels))

    # add one shape to sequence list for standardizing
    new_shape = list(sequence.shape)
    new_shape.append(1)
    sequence = np.reshape(sequence, new_shape)

    # return the dataset with sequence list and labels
    return tf.data.Dataset.from_tensor_slices((sequence, labels))

# Parse input record to features and labels.
def parse(residue, label):
    # convert input to float
    residue = tf.to_float(residue)
    # convert lable to int
    label = tf.to_int64(label)
    # return feature and label
    return {"residue": residue}, {"label": label}
```

- Module gồm các hàm phục vụ cho việc khởi tạo mô hình mạng CNN và hàm tính toán số liệu:

```
# cnn.py

# Import dependences for python 2.7
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf

# Get addition paramesters of model
def get_params():
    return {
        "drop_rate": 0.5
    }

# Define the CNN classifier model
def model(features, labels, mode, params):
    # get sequence of input features
    sequence = features["residue"]
    # get labels of input features
    labels = labels["label"]

    # set drop-out rate equal 0 if it's not traning phase
    drop_rate = params.drop_rate if mode == tf.estimator.ModeKeys.TRAIN else 0.0
    # set training variable to check if it's training phase
    training = mode == tf.estimator.ModeKeys.TRAIN

    # features initialization
    features = sequence
    # loop through all the convolution layer and add related stuff to the model
    for i, filters in enumerate([128]):
        # convolution layer with ReLU activation function
        # and size of filters*kernel_size (128*3 in this case)
        features = tf.layers.conv2d(
```

```
        features, filters=filters, activation=tf.nn.relu, kernel_size=3, padding="same",
        name="conv_%d" % (i + 1))
    # batch-norm layer
    features = tf.layers.batch_normalization(features, training=training)
    # max-pooling layer
    features = tf.layers.max_pooling2d(
        inputs=features, pool_size=2, strides=2, padding="same",
        name="pool_%d" % (i + 1))

# flatten layer
features = tf.contrib.layers.flatten(features)

# drop-out layer
features = tf.layers.dropout(features, drop_rate)
# dense layer with ReLU activation function
features = tf.layers.dense(features, 32, activation=tf.nn.relu, name="dense_1")
# batch-norm layer
features = tf.layers.batch_normalization(features, training=training)

# drop-out layer
features = tf.layers.dropout(features, drop_rate)
# dense layer (final layer) with 2 neurons, has shape [batch_size, 2]
logits = tf.layers.dense(features, params.num_classes,
                         name="dense_2")

# get the predictions
predictions = tf.argmax(logits, axis=1)

# get the cross-entropy loss
loss = tf.losses.sparse_softmax_cross_entropy(
    labels=labels, logits=logits)

# return the result
return {"predictions": predictions}, loss
```

```
# Eval metrics
def eval_metrics(unused_params):
    return {
        # get the accuracy
        "accuracy": tf.contrib.learn.MetricSpec(tf.metrics.accuracy)
    }
```

- Huấn luyện mô hình sử dụng tensorflow:

```
# train_gpu.py

"""This module handles training and evaluation of a neural network model.

Invoke the following command to train the model for one of 5 Fold:
python train_gpu.py --cross_val_index={{fold-index}}

Monitor the logs on Tensorboard:
tensorboard --logdir=output"""

# Import dependences for python 2.7
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
# import cnn module
import cnn
# import pdna mudule
import pdna

# import tensorflow
import tensorflow as tf

# logging for visualizing training process
tf.logging.set_verbosity(tf.logging.INFO)
```

```
# Define the global arguments:
# Model name
tf.flags.DEFINE_string("model", "cnn", "Model name.")
# Dataset name
tf.flags.DEFINE_string("dataset", "pdna", "Dataset name.")
# Optional output dir
tf.flags.DEFINE_string("output_dir", "", "Optional output dir.")
# Schedule
tf.flags.DEFINE_string("schedule", "train_and_evaluate", "Schedule.")
# Hyper parameters
tf.flags.DEFINE_string("hparams", "", "Hyper parameters.")
# Number of training epochs
tf.flags.DEFINE_integer("num_epochs", 20, "Number of training epochs.")
# Summary steps
tf.flags.DEFINE_integer("save_summary_steps", 500, "Summary steps.")
# Checkpoint steps for saving
tf.flags.DEFINE_integer("save_checkpoints_steps", 1000, "Checkpoint steps.")
# Number of eval steps
tf.flags.DEFINE_integer("eval_steps", None, "Number of eval steps.")
# Eval frequency
tf.flags.DEFINE_integer("eval_frequency", 1000, "Eval frequency.")
# Cross validation index
tf.flags.DEFINE_integer("cross_val_index", 0, "Cross validation index.")
# Check for testing phase
tf.flags.DEFINE_boolean("is_test", False, "Check for testing phase.")

# get the global arguments
FLAGS = tf.flags.FLAGS

MODELS = {
    # This is a dictionary of models, the keys are model names, and the values
    # are the module containing get_params, model, and eval_metrics.
    "cnn": cnn
```

```
}

DATASETS = {
    # This is a dictionary of datasets, the keys are dataset names, and the
    # values are the module containing get_params, prepare, read, and parse.
    "pdna": pdna
}

HPARAMS = {
    # optimization option
    "optimizer": "Adam",
    # learning rate
    "learning_rate": 0.00001,
    # number of steps to decay learning rate
    "decay_steps": 5000,
    # batch size
    "batch_size": 128
}

# Aggregates and returns hyper parameters
def get_params():
    # assign hyper parameters
    hparams = HPARAMS
    # add dataset parameters
    hparams.update(DATASETS[FLAGS.dataset].get_params())
    # add model parameters
    hparams.update(MODELS[FLAGS.model].get_params())

    # convert to tf.contrib.training.HParams
    hparams = tf.contrib.training.HParams(**hparams)
    # parse to FLAGS
    hparams.parse(FLAGS.hparams)

    return hparams
```

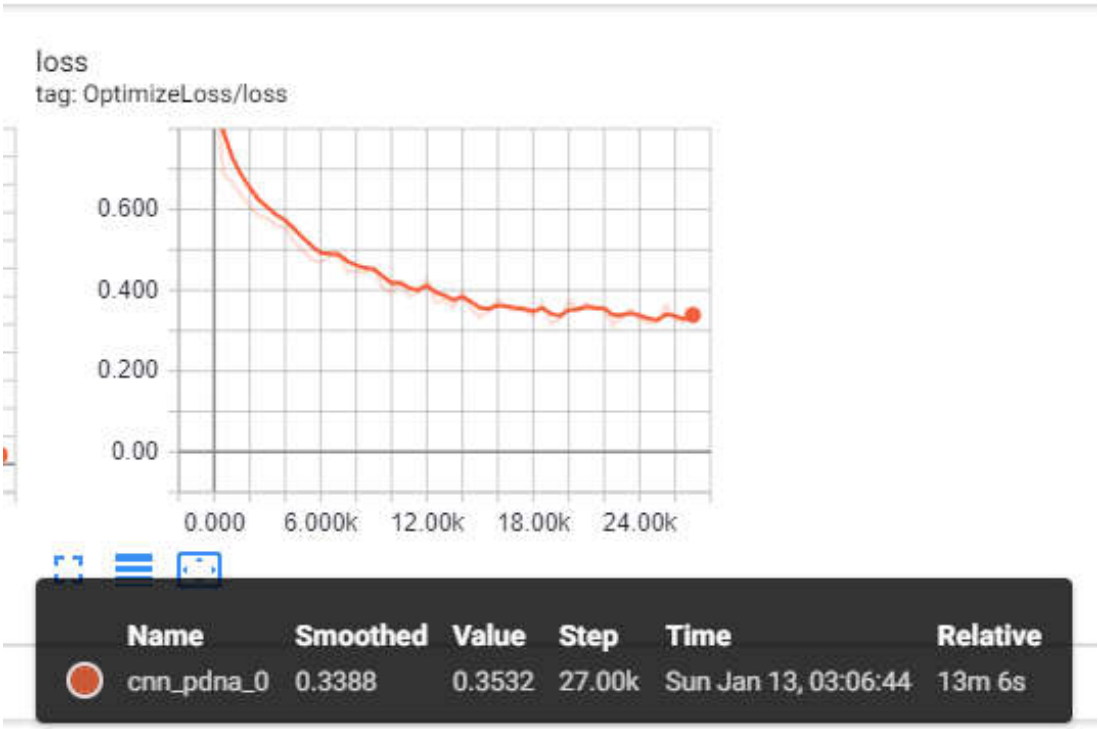
```
# Returns an input function to read the dataset
def make_input_fn(mode, params):
   def _input_fn():
      # get dataset
      dataset = DATASETS[FLAGS.dataset].read(mode, FLAGS.is_test,
FLAGS.cross_val_index)
      # refactor the dataset following number of epochs and batch size
      if mode == tf.estimator.ModeKeys.TRAIN:
         dataset = dataset.repeat(FLAGS.num_epochs)
         dataset = dataset.shuffle(params.batch_size * 5)
      # convert to dataset object
      dataset = dataset.map(
         DATASETS[FLAGS.dataset].parse, num_parallel_calls=8)
      # add the batch size
      dataset = dataset.batch(params.batch_size)
      # creates an iterator for enumerating the elements of dataset.
      iterator = dataset.make_one_shot_iterator()
      features, labels = iterator.get_next()
      return features, labels
   return _input_fn

# Returns a model function
def make_model_fn():
   def _model_fn(features, labels, mode, params):
      # assign the model function
      model_fn = MODELS[FLAGS.model].model
      # recover step or start new step for traning
      global_step = tf.train.get_or_create_global_step()
      # get predictions and loss
      predictions, loss = model_fn(features, labels, mode, params)

      # init training optimization setting
      train_op = None
```

```
        # in the training phase:
        if mode == tf.estimator.ModeKeys.TRAIN:
            # get the decay function following the learning rate and decay steps
            def _decay(learning_rate, global_step):
                learning_rate = tf.train.exponential_decay(
                    learning_rate, global_step, params.decay_steps, 0.5,
                    staircase=True)
                return learning_rate

            # assign the training optimization setting
            train_op = tf.contrib.layers.optimize_loss(
                loss=loss,
                global_step=global_step,
                learning_rate=params.learning_rate,
                optimizer=params.optimizer,
                learning_rate_decay_fn=_decay)

        # return a ModelFnOps instance
        return tf.contrib.learn.ModelFnOps(
            mode=mode,
            predictions=predictions,
            loss=loss,
            train_op=train_op)

    return _model_fn

# Constructs an experiment object
def experiment_fn(run_config, hparams):
    # create the estimator
    estimator = tf.contrib.learn.Estimator(
        model_fn=make_model_fn(), config=run_config, params=hparams)
    # return an Experiment instance
    return tf.contrib.learn.Experiment(
        estimator=estimator,
```

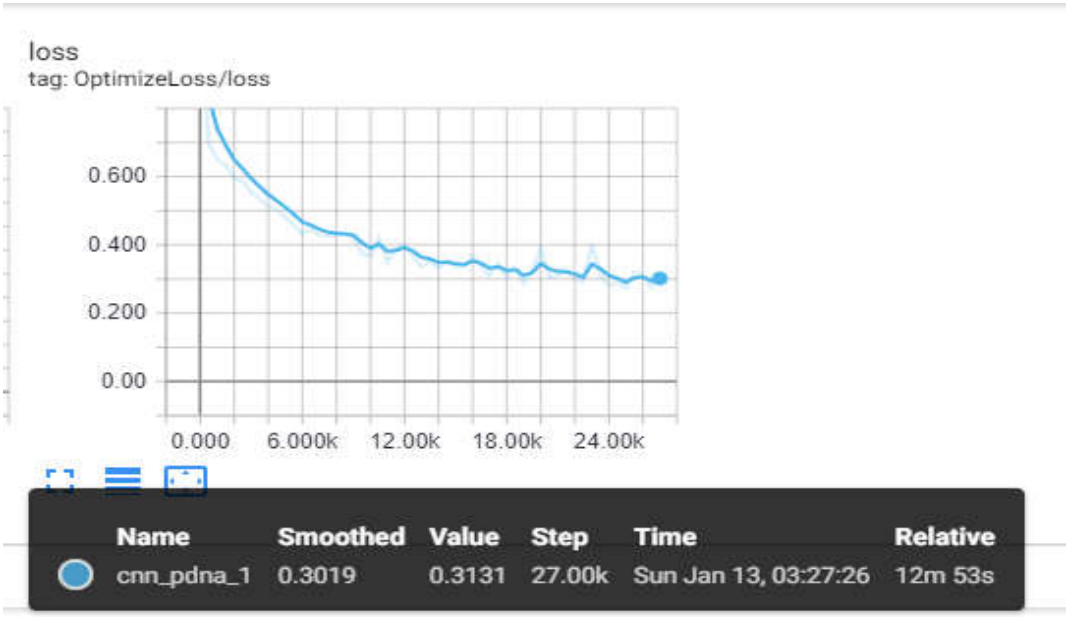
```
        train_input_fn=make_input_fn(tf.estimator.ModeKeys.TRAIN, hparams),
        eval_input_fn=make_input_fn(tf.estimator.ModeKeys.EVAL, hparams),
        eval_metrics=MODELS[FLAGS.model].eval_metrics(hparams),
        eval_steps=FLAGS.eval_steps,
        min_eval_frequency=FLAGS.eval_frequency)

# Main entry point
def main(unused_argv):
    # set the model directory following the ouput_dir
    if FLAGS.output_dir:
        model_dir = FLAGS.output_dir
    # if the ouput_dir is not set, set the model directory following model name, dataset and
cross validation index
    else:
        model_dir = "output/%s_%s_%s" % (FLAGS.model, FLAGS.dataset,
FLAGS.cross_val_index)

    # get dataset prepared
    DATASETS[FLAGS.dataset].prepare()

    # init the session configuration
    session_config = tf.ConfigProto()
    # allow TensorFlow to automatically choose an existing and supported device
    # to run the operations in case the specified one doesn't exist
    session_config.allow_soft_placement = True
    # allow TensorFlow use all the avilable resources of GPU
    session_config.gpu_options.allow_growth = True
    # create a RunConfig instance
    run_config = tf.contrib.learn.RunConfig(
        model_dir=model_dir,
        save_summary_steps=FLAGS.save_summary_steps,
        save_checkpoints_steps=FLAGS.save_checkpoints_steps,
        save_checkpoints_secs=None,
        session_config=session_config)
```

```
    # run the learning process
    tf.contrib.learn.learn_runner.run(
        experiment_fn=experiment_fn,
        run_config=run_config,
        schedule=FLAGS.schedule,
        hparams=get_params())

# if this file run from command line (is the main program), run the app
if __name__ == "__main__":
    tf.app.run()
```

- Đồ thị hàm loss trên tập Train:



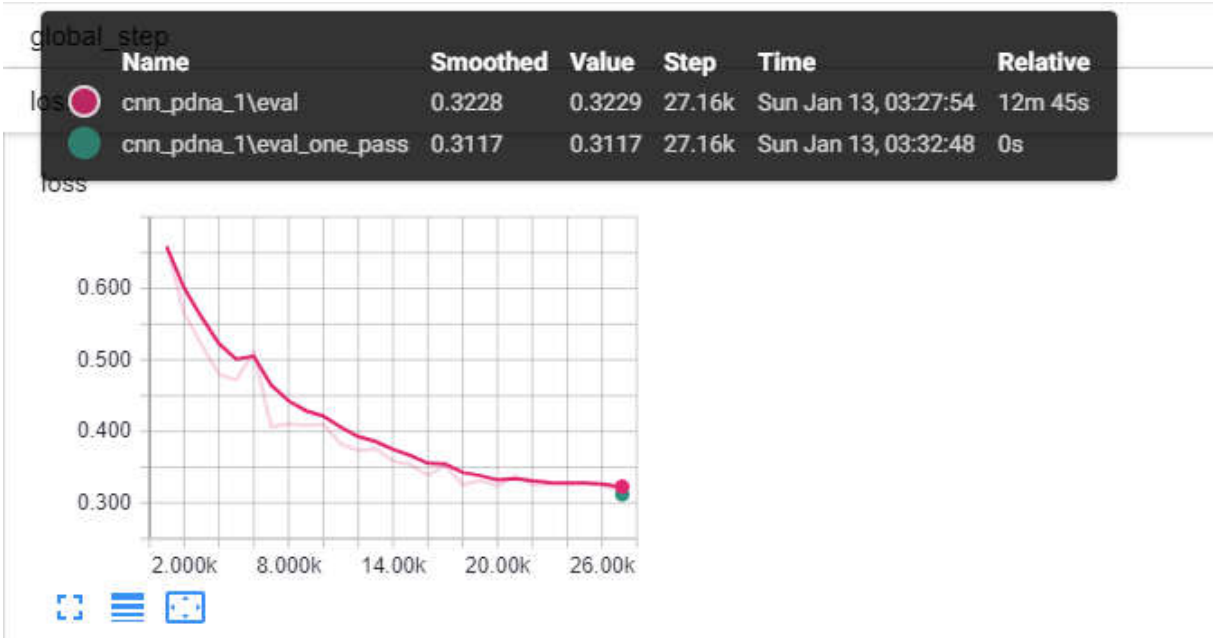
Hình 5: Đồ thị hàm Softmax Cross Entropy Loss khi huấn luyện mô hình trên tập Train (Lần phân chia đầu trong trong 5-folds cross validation của seed 1)



Hình 6: Đồ thị hàm Softmax Cross Entropy Loss khi huấn luyện mô hình trên tập Train (Lần phân chia thứ hai trong trong 5-folds cross validation của seed 1)

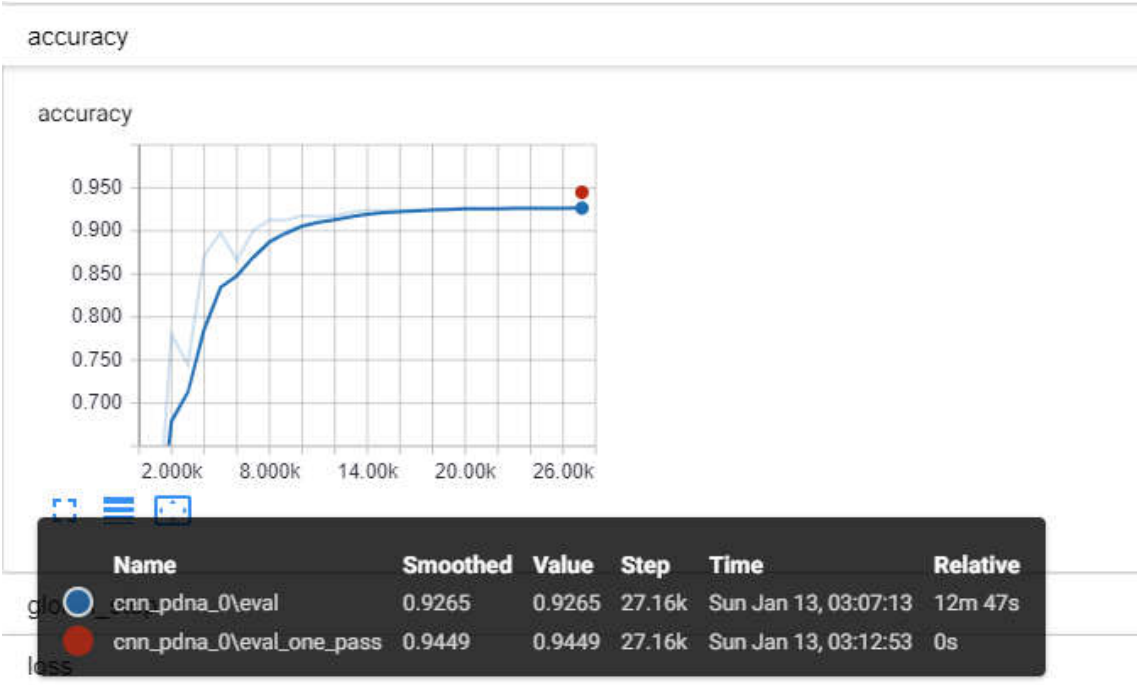
- Đồ thị hàm loss trên tập Validation:

Hình 7: Đồ thị hàm Softmax Cross Entropy Loss trên tập Validation khi huấn luyện mô hình (Lần phân chia đầu trong trong 5-folds cross validation của seed 1)

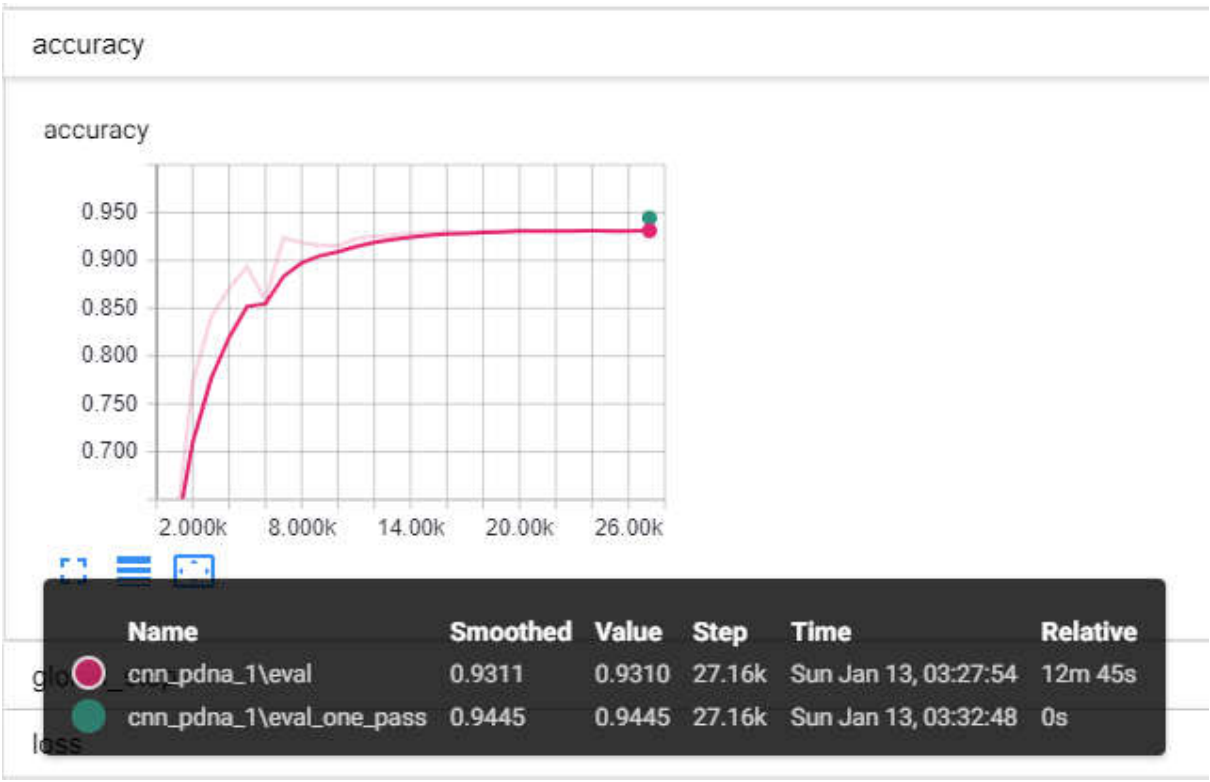


Hình 8: Đồ thị hàm Softmax Cross Entropy Loss trên tập Validation khi huấn luyện mô hình (Lần phân chia thứ hai trong trong 5-folds cross validation của seed 1)

- Đồ thị Accuracy trên tập Validation:



Hình 9: Đồ thị Accuracy trên tập Validation khi huấn luyện mô hình(Lần phân chia đầu trong trong 5-folds cross validation của seed 1)



Hình 10: Đồ thị Accuracy trên tập Validation khi huấn luyện mô hình (Lần phân chia thứ hai trong trong 5-folds cross validation của seed 1)

- Kết quả accuracy (%) cao nhất trên tập Validation khi chạy 10 random seed khác nhau:

	1	2	3	4	5	6	7	8	9	10	**MAX**
Fold 1	92.93	92.73	92.76	92.44	92.75	**93.00**	92.72	92.68	91.92	92.87	**93.00**
Fold 2	93.33	93.03	93.16	93.33	93.43	93.41	**93.45**	93.19	93.39	92.44	**93.45**
Fold 3	**93.28**	92.84	93.17	92.90	92.99	92.80	93.19	93.14	93.01	92.71	**93.28**
Fold 4	92.26	92.44	92.73	92.59	92.61	92.71	**92.82**	92.69	92.23	92.31	**92.82**
Fold 5	92.49	92.55	92.51	92.52	**92.56**	92.28	92.48	92.42	92.11	92.17	**92.56**
MAX											**93.45**

Bảng 4: Kết quả accuracy trên tập Validation khi huấn luyện mô hình với đặc và chạy 10 random seeds khác nhau với 5 Folds

- Kết quả accuracy (%) cao nhất trên tập Test khi chạy 10 random seed khác nhau:

	1	2	3	4	5	6	7	8	9	10	**MAX**
Fold 1	**94.64**	94.42	94.44	93.95	94.63	94.68	94.37	94.49	94.44	94.12	**94.64**
Fold 2	94.54	94.24	94.25	94.63	**94.81**	94.69	94.58	94.45	94.33	94.23	**94.81**
Fold 3	**94.71**	94.14	94.48	94.16	94.47	94.26	94.64	94.27	94.12	94.41	**94.71**
Fold 4	94.06	94.29	94.57	94.46	94.57	94.71	**94.78**	94.60	94.36	94.15	**94.78**
Fold 5	94.48	94.77	94.79	94.82	94.85	94.52	**94.86**	94.58	94.51	94.49	**94.86**
MAX											**94.86**

Bảng 5: Kết quả accuracy trên tập Test từ 50 mô hình đã huấn luyện ở trên (10 random seeds khác nhau với 5 Folds)

- Sau mỗi 1000 steps ≈ 1 epoch, mô hình sẽ được đánh giá trên tập Validation một lần để kiểm tra accuracy hiện tại, nếu accuracy cao hơn giá trị trước đó thì mô hình được lưu lại là mô hình tốt hơn. Sau 20 epochs, mô hình tốt nhất sẽ được sử dụng để kiểm tra trên tập Test
- Quá trình trên được lặp lại trên các Folds mỗi fold 10 lần để tìm ra một mô hình tối ưu nhất.
- Kết quả thu được với accuracy cao nhất cho tập test là 94.86%, cao hơn với kết quả state-of-art là 90.89%

KẾT LUẬN

Việc áp dụng mạng Neural đã cho thấy kết quả khá tốt đối với bài toán dự đoán dư lượng liên kết Protein-DNA. Để có được kết quả này, nhóm em đã phân tích tìm hiểu về tập dataset và cách biểu diễn data để có thể đưa vào áp dụng trong việc huấn luyện mô hình. Nhóm em cũng đã ghi lại các thông tin về hàm loss, accuracy trên tập validation trong quá trình huấn luyện đối với từng lần huấn huyện, và ghi lại độ chính xác trên từng model thu được của tập Test PDNA-Test. Để có thể đạt kết quả tốt hơn nữa, nhóm em sẽ cần thêm thời gian để nghiên cứu và huấn luyện thêm các mô hình khác, kết hợp các mô hình dựa trên một số thuật toán khác như SVM,… và tìm hiểu về cách trích xuất các đặc trưng hiểu quả khác trong việc biểu diễn các dư lượng trong chuỗi protein như PSSM, PSA,…

Qua các bài học trên lớp và qua bài tập lớn lần này, nhóm chúng em đã học được rất nhiều kiến thức quan trọng và căn bản để có thể áp dụng, sử dụng một mô hình mạng neral vào trong một bài toán thực tế. Một lần nữa chúng em xin cảm ơn thầy Nguyễn Hồng Quang đã tận tình chỉ bảo, hướng dẫn để chúng em có thể hoàn thiện được đề tài bài tập lớn lần này!

PHỤ LỤC

- **Bước 1**: Gộp 2 tập data metaDBsite_PDNA316 và PDNA-543 vào thành một file fasta đặt là PDNA-859_sequence.fasta cho tập input và PDNA-859_label.fasta cho tập nhãn.

- **Bước 2**: Đặt data trong đường dẫn "data/TargetDNA/" với folder gốc là fodler của project (chứa các file mã nguồn)

- **Bước 3**: Huấn luyện mô hình:

+ Chạy dòng lệnh:

train_gpu.py --cross_val_index=0

để thực hiện huấn luyện và đánh giá mô hình với lần phân chia dữ liệu train/val đầu tiên của 5-folds cross validation, thể thay đổi các tham số như số epochs, tần suất đánh giá mô hình, số bước lưu checkpoint, …

Sau khi train xong thì chạy dòng lệnh:

train_gpu.py --cross_val_index=0 --schedule=evaluate --is_test

để đánh giá mô hình trên tập test

- **Bước 4**: Lưu mô hình sang một folder khác để có thể sử dụng về sau

- **Bước 5**: Thực hiện 10 lần bước 3 và bước 4 (cho 10 random seeds khác nhau)

- **Bước 6:** Thực hiện lại các bước 3, 4, 5 thêm 4 lần nữa và thay tham số ***--cross_val_index*** lần lượt bằng 1, 2 , 3, 4 cho các lần phân chia dữ liệu train/val còn lại trong 5-folds cross validation.

TÀI LIỆU THAM KHẢO

hu2016.pdf, “Predicting Protein-DNA Binding Residues by Weightedly

Combining Sequence-based Features and Boosting Multiple SVMs”, Jun Hu,

Yang Li, Ming Zhang, Xibei Yang, Hong-Bin Shen, and Dong-Jun Yu,

IEEE/ACM Transactions on Computational Biology

and Bioinformatics, 2016, DOI 10.1109/TCBB.2016.2616469