MoA LSTM

In [1]:

```
TYPE = 'Transformer' # 'LSTM'
```

In [2]:

```
import warnings
warnings.filterwarnings("ignore")

import sys
sys.path.append('../input/iterative-stratification/iterative-stratification-master')
from iterstrat.ml_stratifiers import MultilabelStratifiedKFold

import os
import gc
import datetime
import numpy as np
import pandas as pd
import tensorflow as tf
tf.random.set_seed(42)
import tensorflow.keras.backend as K
import tensorflow.keras.layers as L
import tensorflow.keras.models as M
from tensorflow.keras.callbacks import ReduceLROnPlateau, ModelCheckpoint, EarlyStoppin
g
import tensorflow_addons as tfa
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold, StratifiedKFold
from sklearn.metrics import log_loss
from scipy.optimize import minimize
from tqdm.notebook import tqdm
from time import time

print("Tensorflow version " + tf.__version__)
AUTO = tf.data.experimental.AUTOTUNE
```

```
Tensorflow version 2.4.1
```

In [3]:

```
MIXED_PRECISION = False
XLA_ACCELERATE = True

if MIXED_PRECISION:
    from tensorflow.keras.mixed_precision import experimental as mixed_precision
    if tpu: policy = tf.keras.mixed_precision.experimental.Policy('mixed_bfloat16')
    else: policy = tf.keras.mixed_precision.experimental.Policy('mixed_float16')
    mixed_precision.set_policy(policy)
    print('Mixed precision enabled')

if XLA_ACCELERATE:
    tf.config.optimizer.set_jit(True)
    print('Accelerated Linear Algebra enabled')
```

```
Accelerated Linear Algebra enabled
```

Data Preparation

In [4]:

```
train_features = pd.read_csv('../input/lish-moa/train_features.csv')
train_targets = pd.read_csv('../input/lish-moa/train_targets_scored.csv')
test_features = pd.read_csv('../input/lish-moa/test_features.csv')

ss = pd.read_csv('../input/lish-moa/sample_submission.csv')

cols = [c for c in ss.columns.values if c != 'sig_id']
```

In [5]:

```
def preprocess(df):
    df.loc[:, 'cp_type'] = df.loc[:, 'cp_type'].map({'trt_cp': 0, 'ctl_vehicle': 1})
    df.loc[:, 'cp_dose'] = df.loc[:, 'cp_dose'].map({'D1': 0, 'D2': 1})
    del df['sig_id']
    return df

def log_loss_metric(y_true, y_pred):
    metrics = []
    for _target in range(len(train_targets.columns)):
        metrics.append(log_loss(y_true.values[:, _target], y_pred[:, _target], labels =
[0,1]))
    return np.mean(metrics)

train = preprocess(train_features)
test = preprocess(test_features)

del train_targets['sig_id']
```

In [6]:

```
top_feats = [  0,   1,   2,   3,   5,   6,   8,   9,  10,  11,  12,  14,  15,
        16,  18,  19,  20,  21,  23,  24,  25,  27,  28,  29,  30,  31,
        32,  33,  34,  35,  36,  37,  39,  40,  41,  42,  44,  45,  46,
        48,  50,  51,  52,  53,  54,  55,  56,  57,  58,  59,  60,  61,
        63,  64,  65,  66,  68,  69,  70,  71,  72,  73,  74,  75,  76,
        78,  79,  80,  81,  82,  83,  84,  86,  87,  88,  89,  90,  92,
        93,  94,  95,  96,  97,  99, 100, 101, 103, 104, 105, 106, 107,
       108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120,
       121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 132, 133, 134,
       135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147,
       149, 150, 151, 152, 153, 154, 155, 157, 159, 160, 161, 163, 164,
       165, 166, 167, 168, 169, 170, 172, 173, 175, 176, 177, 178, 180,
       181, 182, 183, 184, 186, 187, 188, 189, 190, 191, 192, 193, 195,
       197, 198, 199, 202, 203, 205, 206, 208, 209, 210, 211, 212, 213,
       214, 215, 218, 219, 220, 221, 222, 224, 225, 227, 228, 229, 230,
       231, 232, 233, 234, 236, 238, 239, 240, 241, 242, 243, 244, 245,
       246, 248, 249, 250, 251, 253, 254, 255, 256, 257, 258, 259, 260,
       261, 263, 265, 266, 268, 270, 271, 272, 273, 275, 276, 277, 279,
       282, 283, 286, 287, 288, 289, 290, 294, 295, 296, 297, 299, 300,
       301, 302, 303, 304, 305, 306, 308, 309, 310, 311, 312, 313, 315,
       316, 317, 320, 321, 322, 324, 325, 326, 327, 328, 329, 330, 331,
       332, 333, 334, 335, 338, 339, 340, 341, 343, 344, 345, 346, 347,
       349, 350, 351, 352, 353, 355, 356, 357, 358, 359, 360, 361, 362,
       363, 364, 365, 366, 368, 369, 370, 371, 372, 374, 375, 376, 377,
       378, 379, 380, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391,
       392, 393, 394, 395, 397, 398, 399, 400, 401, 403, 405, 406, 407,
       408, 410, 411, 412, 413, 414, 415, 417, 418, 419, 420, 421, 422,
       423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435,
       436, 437, 438, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450,
       452, 453, 454, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465,
       466, 468, 469, 471, 472, 473, 474, 475, 476, 477, 478, 479, 482,
       483, 485, 486, 487, 488, 489, 491, 492, 494, 495, 496, 500, 501,
       502, 503, 505, 506, 507, 509, 510, 511, 512, 513, 514, 516, 517,
       518, 519, 521, 523, 525, 526, 527, 528, 529, 530, 531, 532, 533,
       534, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547,
       549, 550, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563,
       564, 565, 566, 567, 569, 570, 571, 572, 573, 574, 575, 577, 580,
       581, 582, 583, 586, 587, 590, 591, 592, 593, 595, 596, 597, 598,
       599, 600, 601, 602, 603, 605, 607, 608, 609, 611, 612, 613, 614,
       615, 616, 617, 619, 622, 623, 625, 627, 630, 631, 632, 633, 634,
       635, 637, 638, 639, 642, 643, 644, 645, 646, 647, 649, 650, 651,
       652, 654, 655, 658, 659, 660, 661, 662, 663, 664, 666, 667, 668,
       669, 670, 672, 674, 675, 676, 677, 678, 680, 681, 682, 684, 685,
       686, 687, 688, 689, 691, 692, 694, 695, 696, 697, 699, 700, 701,
       702, 703, 704, 705, 707, 708, 709, 711, 712, 713, 714, 715, 716,
       717, 723, 725, 727, 728, 729, 730, 731, 732, 734, 736, 737, 738,
       739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751,
       752, 753, 754, 755, 756, 758, 759, 760, 761, 762, 763, 764, 765,
       766, 767, 769, 770, 771, 772, 774, 775, 780, 781, 782, 783, 784,
       785, 787, 788, 790, 793, 795, 797, 799, 800, 801, 805, 808, 809,
       811, 812, 813, 816, 819, 820, 821, 822, 823, 825, 826, 827, 829,
       831, 832, 833, 834, 835, 837, 838, 839, 840, 841, 842, 844, 845,
       846, 847, 848, 850, 851, 852, 854, 855, 856, 858, 860, 861, 862,
       864, 867, 868, 870, 871, 873, 874]
print(len(top_feats))
```

696

Model Functions

Base Transformer structure from https://www.tensorflow.org/tutorials/text/transformer (https://www.tensorflow.org/tutorials/text/transformer), modified with gelu activation function. No positional embedding is needed so I remove it and then changes the embedding layer to dense layer.

In [7]:

```
def scaled_dot_product_attention(q, k, v, mask):
    """Calculate the attention weights.
    q, k, v must have matching leading dimensions.
    k, v must have matching penultimate dimension, i.e.: seq_len_k = seq_len_v.
    The mask has different shapes depending on its type(padding or look ahead)
    but it must be broadcastable for addition.

    Args:
    q: query shape == (..., seq_len_q, depth)
    k: key shape == (..., seq_len_k, depth)
    v: value shape == (..., seq_len_v, depth_v)
    mask: Float tensor with shape broadcastable
          to (..., seq_len_q, seq_len_k). Defaults to None.

    Returns:
    output, attention_weights
    """

    matmul_qk = tf.matmul(q, k, transpose_b = True)  # (..., seq_len_q, seq_len_k)

    # scale matmul_qk
    dk = tf.cast(tf.shape(k)[-1], tf.float32)
    scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

    # add the mask to the scaled tensor.
    if mask is not None:

        scaled_attention_logits += (mask * -1e9)

    # softmax is normalized on the last axis (seq_len_k) so that the scores
    # add up to 1.
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis = -1)  # (..., seq_
len_q, seq_len_k)

    output = tf.matmul(attention_weights, v)  # (..., seq_len_q, depth_v)

    return output, attention_weights

class MultiHeadAttention(tf.keras.layers.Layer):

    def __init__(self, d_model, num_heads):

        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.d_model = d_model

        assert d_model % self.num_heads == 0

        self.depth = d_model // self.num_heads

        self.wq = tf.keras.layers.Dense(d_model)
        self.wk = tf.keras.layers.Dense(d_model)
        self.wv = tf.keras.layers.Dense(d_model)

        self.dense = tf.keras.layers.Dense(d_model)

    def split_heads(self, x, batch_size):
        """Split the last dimension into (num_heads, depth).
        Transpose the result such that the shape is (batch_size, num_heads, seq_len, de
```

```
pth)
        """
        x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
        return tf.transpose(x, perm = [0, 2, 1, 3])

    def call(self, v, k, q, mask):

        batch_size = tf.shape(q)[0]

        q = self.wq(q)  # (batch_size, seq_len, d_model)
        k = self.wk(k)  # (batch_size, seq_len, d_model)
        v = self.wv(v)  # (batch_size, seq_len, d_model)

        q = self.split_heads(q, batch_size)  # (batch_size, num_heads, seq_len_q, dept
h)
        k = self.split_heads(k, batch_size)  # (batch_size, num_heads, seq_len_k, dept
h)
        v = self.split_heads(v, batch_size)  # (batch_size, num_heads, seq_len_v, dept
h)

        # scaled_attention.shape == (batch_size, num_heads, seq_len_q, depth)
        # attention_weights.shape == (batch_size, num_heads, seq_len_q, seq_len_k)
        scaled_attention, attention_weights = scaled_dot_product_attention(
            q, k, v, mask)

        scaled_attention = tf.transpose(scaled_attention, perm = [0, 2, 1, 3])  # (batc
h_size, seq_len_q, num_heads, depth)

        concat_attention = tf.reshape(scaled_attention,
                                      (batch_size, -1, self.d_model))  # (batch_size, s
eq_len_q, d_model)

        output = self.dense(concat_attention)  # (batch_size, seq_len_q, d_model)

        return output, attention_weights

def gelu(x):
    """Gaussian Error Linear Unit.
    This is a smoother version of the RELU.
    Original paper: https://arxiv.org/abs/1606.08415
    refer : https://github.com/google-research/bert/blob/bee6030e31e42a9394ac567da170a8
9a98d2062f/modeling.py#L264
    Args:
        x: float Tensor to perform activation.
    Returns:
        `x` with the GELU activation applied.
    """
    cdf = 0.5 * (1.0 + tf.tanh(
        (np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
    return x * cdf

def point_wise_feed_forward_network(d_model, dff):

    return tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation = gelu),  # (batch_size, seq_len, dff)
      tf.keras.layers.Dense(d_model)  # (batch_size, seq_len, d_model)
    ])

class EncoderLayer(tf.keras.layers.Layer):

    def __init__(self, d_model, num_heads, dff, rate = 0.1):
```

```
        super(EncoderLayer, self).__init__()

        self.mha = MultiHeadAttention(d_model, num_heads)
        self.ffn = point_wise_feed_forward_network(d_model, dff)

        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon = 1e-6)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon = 1e-6)

        self.dropout1 = tf.keras.layers.Dropout(rate)
        self.dropout2 = tf.keras.layers.Dropout(rate)

    def call(self, x, training, mask):

        attn_output, _ = self.mha(x, x, x, mask)  # (batch_size, input_seq_len, d_mode
l)
        attn_output = self.dropout1(attn_output, training = training)
        out1 = self.layernorm1(x + attn_output)  # (batch_size, input_seq_len, d_model)

        ffn_output = self.ffn(out1)  # (batch_size, input_seq_len, d_model)
        ffn_output = self.dropout2(ffn_output, training = training)
        out2 = self.layernorm2(out1 + ffn_output)  # (batch_size, input_seq_len, d_mode
l)

        return out2

class TransformerEncoder(tf.keras.layers.Layer):

    def __init__(self, num_layers, d_model, num_heads, dff, rate = 0.1):

        super(TransformerEncoder, self).__init__()

        self.d_model = d_model
        self.num_layers = num_layers
        self.num_heads = num_heads
        self.dff = dff
        self.rate = rate

        self.embedding = tf.keras.layers.Dense(self.d_model)

        self.enc_layers = [EncoderLayer(self.d_model, self.num_heads, self.dff, self.ra
te)
                           for _ in range(self.num_layers)]

        self.dropout = tf.keras.layers.Dropout(self.rate)

    def get_config(self):

        config = super().get_config().copy()
        config.update({
            'num_layers': self.num_layers,
            'd_model': self.d_model,
            'num_heads': self.num_heads,
            'dff': self.dff,
            'dropout': self.dropout,
        })
        return config

    def call(self, x, training, mask = None):

        seq_len = tf.shape(x)[1]
```

```
        x = self.embedding(x)

        x = self.dropout(x, training = training)

        for i in range(self.num_layers):

            x = self.enc_layers[i](x, training, mask)

        return x  # (batch_size, input_seq_len, d_model)
```

Create Model

In [8]:

```
def create_RNN(num_columns, hidden_units, dropout_rate, learning_rate):

    inp = tf.keras.layers.Input(shape = (num_columns, ))
    x = tf.keras.layers.Reshape((1, num_columns))(inp)

    for i, units in enumerate(hidden_units[:-1]):
        if i == 0:
            x, h, c = tf.keras.layers.LSTM(units, dropout = dropout_rate, return_sequen
ces = True, return_state = True)(x)
        else:
            x, h, c = tf.keras.layers.LSTM(units, dropout = dropout_rate, return_sequen
ces = True, return_state = True)(x, initial_state = [h, c])

    x = tf.keras.layers.LSTM(hidden_units[-1], dropout = dropout_rate)(x, initial_state
= [h, c])

    x = tf.keras.layers.BatchNormalization()(x)

    out = tfa.layers.WeightNormalization(tf.keras.layers.Dense(206, activation = 'sigmo
id'))(x)

    model = tf.keras.models.Model(inputs = inp, outputs = out)

    model.compile(optimizer = tfa.optimizers.Lookahead(tf.optimizers.Adam(learning_rate
), sync_period = 10),
                  loss = 'binary_crossentropy')

    return model
```

In [9]:

```
def create_Transformer(num_columns, num_layers, d_model, num_heads, dff, dropout_rate,
learning_rate):
    # d_model: Embedding depth of the model.
    # num_heads: Number of heads for Multi-head attention. d_model % num_heads = 0
    # dff: Depth of the point wise feed-forward network

    inp = tf.keras.layers.Input(shape = (num_columns, ))
    x = tf.keras.layers.Reshape((1, num_columns))(inp)

    x = TransformerEncoder(num_layers, d_model, num_heads, dff, dropout_rate)(x)[:, 0,
:]

    out = tfa.layers.WeightNormalization(tf.keras.layers.Dense(206, activation = 'sigmo
id'))(x)

    model = tf.keras.models.Model(inputs = inp, outputs = out)

    model.compile(optimizer = tfa.optimizers.Lookahead(tf.optimizers.Adam(learning_rate
), sync_period = 10),
                  loss = 'binary_crossentropy')

    return model
```

Train Model

In [10]:

```
N_STARTS = 3
N_SPLITS = 5

res = train_targets.copy()
ss.loc[:, train_targets.columns] = 0
res.loc[:, train_targets.columns] = 0

for seed in range(N_STARTS):

    for n, (tr, te) in enumerate(MultilabelStratifiedKFold(n_splits = N_SPLITS, random_
state = seed, shuffle = True).split(train_targets, train_targets)):

        start_time = time()
        x_tr, x_val = train.values[tr][:, top_feats], train.values[te][:, top_feats]
        y_tr, y_val = train_targets.astype(float).values[tr], train_targets.astype(floa
t).values[te]
        x_tt = test_features.values[:, top_feats]

        scaler = StandardScaler()
        x_tr = scaler.fit_transform(x_tr)
        x_val = scaler.transform(x_val)
        x_tt = scaler.transform(x_tt)

        if TYPE == 'LSTM':

            model = create_RNN(len(top_feats), [1024, 1024], 0.4, 1e-3)

        elif TYPE == 'Transformer':

            model = create_Transformer(len(top_feats), 3, 128, 8, 256, 0.4, 1e-3)

        rlr = ReduceLROnPlateau(monitor = 'val_loss', factor = 0.1, patience = 3, verbo
se = 0,
                                min_delta = 1e-4, mode = 'min')
        ckp = ModelCheckpoint(f'{TYPE}_{seed}_{n}.hdf5', monitor = 'val_loss', verbose
= 0,
                              save_best_only = True, save_weights_only = True, mode =
'min')
        es = EarlyStopping(monitor = 'val_loss', min_delta = 1e-4, patience = 7, mode =
'min',
                           baseline = None, restore_best_weights = True, verbose = 0)

        model.fit(x_tr, y_tr, validation_data = (x_val, y_val), epochs = 100, batch_siz
e = 128,
                  callbacks = [rlr, ckp, es], verbose = 0)

        model.load_weights(f'{TYPE}_{seed}_{n}.hdf5')
        ss.loc[:, train_targets.columns] += model.predict(x_tt, batch_size = 128) / (N_
SPLITS * N_STARTS)
        fold_pred = model.predict(x_val, batch_size = 128)
        res.loc[te, train_targets.columns] += fold_pred / N_STARTS
        fold_score = log_loss_metric(train_targets.loc[te], fold_pred)
        print(f'[{str(datetime.timedelta(seconds = time() - start_time))[2:7]}] {TYPE}:
Seed {seed}, Fold {n}:', fold_score)

        K.clear_session()
        del model, fold_pred, fold_score
        x = gc.collect()
```

```
[02:23] Transformer: Seed 0, Fold 0: 0.015819130394037855
[02:22] Transformer: Seed 0, Fold 1: 0.015792717676688325
[02:14] Transformer: Seed 0, Fold 2: 0.015740498471066112
[02:17] Transformer: Seed 0, Fold 3: 0.015826392333468196
[02:23] Transformer: Seed 0, Fold 4: 0.015566123613364534
[02:19] Transformer: Seed 1, Fold 0: 0.015727420644164082
[02:26] Transformer: Seed 1, Fold 1: 0.0157092735417686
[02:27] Transformer: Seed 1, Fold 2: 0.015704119687034765
[02:31] Transformer: Seed 1, Fold 3: 0.015723208845577607
[02:25] Transformer: Seed 1, Fold 4: 0.01564101665677647
[02:15] Transformer: Seed 2, Fold 0: 0.015642292272609208
[02:21] Transformer: Seed 2, Fold 1: 0.015700195654116105
[02:22] Transformer: Seed 2, Fold 2: 0.015859273292385612
[02:50] Transformer: Seed 2, Fold 3: 0.015765626329097632
[02:34] Transformer: Seed 2, Fold 4: 0.01566845392634351
```

In [11]:

```
print(f'{TYPE} OOF Metric: {log_loss_metric(train_targets, res.values)}')
res.loc[train['cp_type'] == 1, train_targets.columns] = 0
ss.loc[test['cp_type'] == 1, train_targets.columns] = 0
print(f'{TYPE} OOF Metric with postprocessing: {log_loss_metric(train_targets, res.values)}')
```

```
Transformer OOF Metric: 0.015406901389707536
Transformer OOF Metric with postprocessing: 0.015399527014047896
```

Submit

In [12]:

```
ss.to_csv('./submission.csv', index = False)
```