В этом ноутбуке пробуется модель на основе LSTM (так как у нас тут что-то похожее на временные ряды, то есть гипотеза, что LSTM хорошо здесь зайдет), обученная на предсказаниях леса. Ноутбук вдохновлен вот этим [https://www.kaggle.com/khalildmk/simple-two-layer-bidirectional-lstm-with-pytorch (https://www.kaggle.com/khalildmk/simple-two-layer-bidirectional-lstm-with-pytorch)](https://www.kaggle.com/khalildmk/simple-two-layer-bidirectional-lstm-with-pytorch) - оттуда взято большинство кода, но была переделана работа с данными и размерности в самой модели.

In [0]:

```
import numpy as np
import pandas as pd
import os
import torch
import torch.nn as nn
import time
import copy
from torch.utils.data import Dataset, DataLoader
import torch.nn.functional as F
from sklearn.metrics import f1_score
from sklearn.model_selection import KFold
device = torch.device("cuda:0") if torch.cuda.is_available() else torch.devi
from torch.optim.lr_scheduler import ReduceLROnPlateau
import gc
```

1. Параметры и модель

In [0]:

```
n_epochs = 100
lr = 0.01
n_folds = 5
lstm_input_size = 12
hidden_state_size = 30
batch_size = 30
num_sequence_layers = 2
output_dim = 11
num_time_steps = 4000
rnn_type = 'LSTM'
```

In [0]:

```
class Bi_RNN(nn.Module):

    def __init__(self, input_dim, hidden_dim, batch_size, output_dim=11, num
        super(Bi_RNN, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.batch_size = batch_size
        self.num_layers = num_layers

        #Define the initial linear hidden layer
        self.init_linear = nn.Linear(self.input_dim, self.input_dim)

        # Define the LSTM layer
        self.lstm = eval('nn.' + rnn_type)(self.input_dim, self.hidden_dim,

        # Define the output layer
        self.linear = nn.Linear(self.hidden_dim * 2, output_dim)

    def init_hidden(self):
        # This is what we'll initialise our hidden state as
        return (torch.zeros(self.num_layers, self.batch_size, self.hidden_di
                torch.zeros(self.num_layers, self.batch_size, self.hidden_di

    def forward(self, input):
        #Forward pass through initial hidden layer
        linear_input = self.init_linear(input)

        # Forward pass through LSTM layer
        # shape of lstm_out: [batch_size, input_size ,hidden_dim]
        # shape of self.hidden: (a, b), where a and b both
        # have shape (batch_size, num_layers, hidden_dim).
        lstm_out, self.hidden = self.lstm(linear_input)

        # Can pass on the entirety of lstm_out to the next layer if it is a
        y_pred = self.linear(lstm_out)
        return y_pred
```

2. Даталоадеры

In [0]:

```
class ION_Dataset_Sequential(Dataset):
    def __init__(self, input, output):
        self.input = input
        self.output = output

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

    def __getitem__(self, idx):
        x = self.input[idx]
        y = self.output[idx]
        x = torch.tensor(x, dtype=torch.float)
        y = torch.tensor(y, dtype=torch.float)
        return x, y

class ION_Dataset_Sequential_test(Dataset):
    def __init__(self, input):
        self.input = input

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

    def __getitem__(self, idx):
        x = self.input[idx]
        x = torch.tensor(x, dtype=torch.float)
        return x
```

3. Данные

In [5]:

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

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call
drive.mount("/content/gdrive", force_remount=True).
```

In [0]:

```
train_df = pd.read_csv('/content/gdrive/My Drive/data/train_clean.csv')
test_df = pd.read_csv('/content/gdrive/My Drive/data/test_clean.csv')
```

In [0]:

```
train_probs = np.load('/content/gdrive/My Drive/data/Y_train_proba.npy')
test_probs = np.load('/content/gdrive/My Drive/data/Y_test_proba.npy')
```

In [0]:

```
train_df = pd.concat([train_df, pd.DataFrame(train_probs)], axis=1)
test_df = pd.concat([test_df, pd.DataFrame(test_probs)], axis=1)
```

In [0]:

```
train_df.columns = train_df.columns.astype(str)
test_df.columns = test_df.columns.astype(str)
```

In [0]:
```
X = train_df[['signal', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '1
y = pd.get_dummies(train_df['open_channels']).values.reshape(-1, num_time_st
test_input = test_df[['signal', '0', '1', '2', '3', '4', '5', '6', '7', '8',
train_input_mean = X.mean()
train_input_sigma = X.std()
test_input = (test_input-train_input_mean)/train_input_sigma
test_preds = np.zeros((int(test_input.shape[0] * test_input.shape[1])))
test = ION_Dataset_Sequential_test(test_input)
test_loader = DataLoader(test, batch_size=batch_size, shuffle=False)
```

4. Обучение с разбиением на фолды

In [11]:

```python
kfold = KFold(n_splits=n_folds, shuffle=True, random_state=42)
local_val_score = 0
models = {}

k=0 #initialize fold number
for tr_idx, val_idx in kfold.split(X, y):
    test_p = np.zeros((int(test_input.shape[0] * test_input.shape[1])))

    print('starting fold', k)
    k += 1

    print(6*'#', 'splitting and reshaping the data')
    train_input = X[tr_idx]
    print(train_input.shape)
    train_target = y[tr_idx]
    val_input = X[val_idx]
    val_target = y[val_idx]
    train_input_mean = train_input.mean()
    train_input_sigma = train_input.std()
    val_input = (val_input-train_input_mean)/train_input_sigma
    train_input = (train_input-train_input_mean)/train_input_sigma

    print(6*'#', 'Loading')
    train = ION_Dataset_Sequential(train_input, train_target)
    valid = ION_Dataset_Sequential(val_input, val_target)
    train_loader = torch.utils.data.DataLoader(train, batch_size=batch_size,
    valid_loader = torch.utils.data.DataLoader(valid, batch_size=batch_size,

    #Build tensor data for torch
    train_preds = np.zeros((int(train_input.shape[0] * train_input.shape[1])
    val_preds = np.zeros((int(val_input.shape[0] * val_input.shape[1])))
    best_val_preds = np.zeros((int(val_input.shape[0] * val_input.shape[1]))
    train_targets = np.zeros((int(train_input.shape[0] * train_input.shape[
    avg_losses_f = []
    avg_val_losses_f = []

    #Define loss function
    loss_fn = torch.nn.BCEWithLogitsLoss()

    #Build model, initialize weights and define optimizer
    model = Bi_RNN(lstm_input_size, hidden_state_size, batch_size=batch_size
    model = model.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=1e
    scheduler = ReduceLROnPlateau(optimizer, 'min', patience=150, factor=0.
    temp_val_loss = 9999999999
    reached_val_score = 0

    #Iterate through epochs
    for epoch in range(n_epochs):
        start_time = time.time()

        #Train
        model.train()
        avg_loss = 0.
        for i, (x_batch, y_batch) in enumerate(train_loader):
            x_batch = x_batch.view(-1, num_time_steps, lstm_input_size)
```

```
            y_batch = y_batch.view(-1, num_time_steps, output_dim)
            optimizer.zero_grad()
            y_pred = model(x_batch.cuda())
            loss = loss_fn(y_pred.cpu(), y_batch)
            loss.backward()
            optimizer.step()
            avg_loss += loss.item() / len(train_loader)
            pred = F.softmax(y_pred, 2).detach().cpu().numpy().argmax(axis=
            train_preds[i * batch_size * train_input.shape[1]:(i + 1) * batc
            train_targets[i * batch_size * train_input.shape[1]:(i + 1) * ba
            del y_pred, loss, x_batch, y_batch, pred

        #Evaluate
        model.eval()
        avg_val_loss = 0.
        for i, (x_batch, y_batch) in enumerate(valid_loader):
            x_batch = x_batch.view(-1, num_time_steps, lstm_input_size)
            y_batch = y_batch.view(-1, num_time_steps, output_dim)
            y_pred = model(x_batch.cuda()).detach()
            avg_val_loss += loss_fn(y_pred.cpu(), y_batch).item() / len(vali
            pred = F.softmax(y_pred, 2).detach().cpu().numpy().argmax(axis=
            val_preds[i * batch_size * val_input.shape[1]:(i + 1) * batch_si
            del y_pred, x_batch, y_batch, pred
        if avg_val_loss < temp_val_loss:
            temp_val_loss = avg_val_loss

        #Calculate F1-score
        train_score = f1_score(train_targets, train_preds, average='macro')
        val_score = f1_score(val_target.argmax(axis=2).reshape((-1)), val_pr

        #Print output of epoch
        elapsed_time = time.time() - start_time
        scheduler.step(avg_val_loss)
        if epoch%10 == 0:
            print('Epoch {}/{} \t loss={:.4f} \t train_f1={:.4f} \t val_loss

        if val_score > reached_val_score:
            reached_val_score = val_score
            best_model = copy.deepcopy(model.state_dict())
            best_val_preds = copy.deepcopy(val_preds)

    #Calculate F1-score of the fold
    val_score_fold = f1_score(val_target.argmax(axis=2).reshape((-1)), best_

    #Save the fold's model in a dictionary
    models[k] = best_model

    #Print F1-score of the fold
    print("BEST VALIDATION SCORE (F1): ", val_score_fold)
    local_val_score += (1/n_folds) * val_score_fold

#Print final average k-fold CV F1-score
print("Final Score ", local_val_score)
```

```
starting fold 0
###### splitting and reshaping the data
```

```
(1000, 4000, 12)
###### Loading
Epoch 1/100      loss=0.2928     train_f1=0.1054         val_loss=0.2055
val_f1=0.2158    time=41.63s
Epoch 11/100     loss=0.0204     train_f1=0.8489         val_loss=0.0238
val_f1=0.8492    time=42.01s
Epoch 21/100     loss=0.0165     train_f1=0.9382         val_loss=0.0193
val_f1=0.9373    time=41.90s
Epoch 31/100     loss=0.0164     train_f1=0.9377         val_loss=0.0188
val_f1=0.9372    time=41.97s
Epoch 41/100     loss=0.0156     train_f1=0.9384         val_loss=0.0185
val_f1=0.9372    time=42.19s
Epoch 51/100     loss=0.0157     train_f1=0.9382         val_loss=0.0185
val_f1=0.9374    time=41.84s
Epoch 61/100     loss=0.0162     train_f1=0.9379         val_loss=0.0190
val_f1=0.9371    time=42.07s
Epoch 71/100     loss=0.0157     train_f1=0.9383         val_loss=0.0186
val_f1=0.9375    time=41.90s
Epoch 81/100     loss=0.0156     train_f1=0.9383         val_loss=0.0184
val_f1=0.9376    time=42.10s
Epoch 91/100     loss=0.0152     train_f1=0.9383         val_loss=0.0184
val_f1=0.9375    time=41.94s
BEST VALIDATION SCORE (F1):  0.9377366523157712
starting fold 1
###### splitting and reshaping the data
(1000, 4000, 12)
###### Loading
Epoch 1/100      loss=0.2628     train_f1=0.1831         val_loss=0.1796
val_f1=0.2677    time=41.80s
Epoch 11/100     loss=0.0209     train_f1=0.8492         val_loss=0.0259
val_f1=0.8491    time=41.80s
Epoch 21/100     loss=0.0168     train_f1=0.9378         val_loss=0.0208
val_f1=0.9368    time=41.91s
Epoch 31/100     loss=0.0156     train_f1=0.9381         val_loss=0.0198
val_f1=0.9371    time=41.90s
Epoch 41/100     loss=0.0159     train_f1=0.9379         val_loss=0.0197
val_f1=0.9371    time=41.70s
Epoch 51/100     loss=0.0154     train_f1=0.9382         val_loss=0.0199
val_f1=0.9359    time=41.60s
Epoch 61/100     loss=0.0151     train_f1=0.9383         val_loss=0.0196
val_f1=0.9368    time=41.82s
Epoch 71/100     loss=0.0165     train_f1=0.9370         val_loss=0.0200
val_f1=0.9369    time=41.43s
Epoch 81/100     loss=0.0152     train_f1=0.9381         val_loss=0.0193
val_f1=0.9372    time=41.68s
Epoch 91/100     loss=0.0153     train_f1=0.9382         val_loss=0.0194
val_f1=0.9371    time=41.30s
BEST VALIDATION SCORE (F1):  0.9376231785584731
starting fold 2
###### splitting and reshaping the data
(1000, 4000, 12)
###### Loading
Epoch 1/100      loss=0.2652     train_f1=0.1949         val_loss=0.1760
val_f1=0.2983    time=41.88s
Epoch 11/100     loss=0.0225     train_f1=0.8494         val_loss=0.0252
val_f1=0.8475    time=41.57s
Epoch 21/100     loss=0.0170     train_f1=0.9378         val_loss=0.0192
```

```
val_f1=0.9370    time=40.97s
Epoch 31/100     loss=0.0167     train_f1=0.9367         val_loss=0.0193
val_f1=0.9366    time=40.41s
Epoch 41/100     loss=0.0162     train_f1=0.9379         val_loss=0.0182
val_f1=0.9375    time=40.79s
Epoch 51/100     loss=0.0156     train_f1=0.9381         val_loss=0.0181
val_f1=0.9375    time=40.29s
Epoch 61/100     loss=0.0157     train_f1=0.9381         val_loss=0.0179
val_f1=0.9377    time=40.46s
Epoch 71/100     loss=0.0156     train_f1=0.9379         val_loss=0.0178
val_f1=0.9377    time=40.37s
Epoch 81/100     loss=0.0158     train_f1=0.9374         val_loss=0.0182
val_f1=0.9369    time=40.44s
Epoch 91/100     loss=0.0158     train_f1=0.9380         val_loss=0.0185
val_f1=0.9368    time=40.86s
BEST VALIDATION SCORE (F1):  0.9380991640612855
starting fold 3
###### splitting and reshaping the data
(1000, 4000, 12)
###### Loading
Epoch 1/100      loss=0.2821     train_f1=0.1583         val_loss=0.1878
val_f1=0.2488    time=40.29s
Epoch 11/100     loss=0.0225     train_f1=0.8489         val_loss=0.0228
val_f1=0.8505    time=40.34s
Epoch 21/100     loss=0.0174     train_f1=0.9378         val_loss=0.0177
val_f1=0.9395    time=40.54s
Epoch 31/100     loss=0.0162     train_f1=0.9375         val_loss=0.0172
val_f1=0.9388    time=40.92s
Epoch 41/100     loss=0.0161     train_f1=0.9375         val_loss=0.0175
val_f1=0.9390    time=40.51s
Epoch 51/100     loss=0.0159     train_f1=0.9378         val_loss=0.0166
val_f1=0.9396    time=40.60s
Epoch 61/100     loss=0.0160     train_f1=0.9378         val_loss=0.0165
val_f1=0.9396    time=40.51s
Epoch 71/100     loss=0.0160     train_f1=0.9375         val_loss=0.0164
val_f1=0.9394    time=40.49s
Epoch 81/100     loss=0.0157     train_f1=0.9378         val_loss=0.0165
val_f1=0.9396    time=40.96s
Epoch 91/100     loss=0.0159     train_f1=0.9379         val_loss=0.0165
val_f1=0.9398    time=40.47s
BEST VALIDATION SCORE (F1):  0.93980716248632
starting fold 4
###### splitting and reshaping the data
(1000, 4000, 12)
###### Loading
Epoch 1/100      loss=0.2710     train_f1=0.1693         val_loss=0.1904
val_f1=0.2708    time=40.36s
Epoch 11/100     loss=0.0208     train_f1=0.8489         val_loss=0.0254
val_f1=0.8496    time=40.46s
Epoch 21/100     loss=0.0159     train_f1=0.9380         val_loss=0.0203
val_f1=0.9375    time=41.19s
Epoch 31/100     loss=0.0155     train_f1=0.9381         val_loss=0.0199
val_f1=0.9379    time=40.58s
Epoch 41/100     loss=0.0153     train_f1=0.9383         val_loss=0.0196
val_f1=0.9378    time=40.38s
Epoch 51/100     loss=0.0154     train_f1=0.9379         val_loss=0.0194
val_f1=0.9380    time=40.27s
```

```
Epoch 61/100     loss=0.0152     train_f1=0.9383         val_loss=0.0195
val_f1=0.9382    time=40.18s
Epoch 71/100     loss=0.0151     train_f1=0.9379         val_loss=0.0199
val_f1=0.9377    time=40.22s
Epoch 81/100     loss=0.0153     train_f1=0.9379         val_loss=0.0194
val_f1=0.9382    time=40.19s
Epoch 91/100     loss=0.0151     train_f1=0.9378         val_loss=0.0196
val_f1=0.9375    time=40.18s
BEST VALIDATION SCORE (F1):  0.9384948498276348
Final Score  0.938352201449897
```

5. Предсказание - усреднение предсказаний моделей, обученных на разных фолдах

In [0]:
```
for k in range(n_folds):
    test_p = np.zeros((int(test_input.shape[0] * test_input.shape[1])))
    k += 1

    #Import model of fold k
    model = Bi_RNN(lstm_input_size, hidden_state_size, batch_size=batch_size
    model = model.to(device)
    model.load_state_dict(models[k])

    #Make predictions on test data
    model.eval()
    for i, x_batch in enumerate(test_loader):
        x_batch = x_batch.view(-1, num_time_steps, lstm_input_size)
        y_pred = model(x_batch.cuda()).detach()
        pred = F.softmax(y_pred, 2).detach().cpu().numpy().argmax(axis=-1)
        test_p[i * batch_size * test_input.shape[1]:(i + 1) * batch_size * t
        del y_pred, x_batch, pred
    test_preds += (1/n_folds) * test_p
```

6. Сохраняем ответы в файл

In [0]:
```
df_sub = pd.read_csv("/content/gdrive/My Drive/data/sample_submission.csv",
df_sub.open_channels = np.array(test_preds, np.int)
df_sub.to_csv("/content/gdrive/My Drive/data/submission_bilstm.csv", index=F
```

Результат: 0.939 на public lb