Задача № 2

по курсу "Нейронные сети: задачи и вычисления"

студент: Бочкарев Фёдор Сергеевич

Загрузка библиотек

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization, Dropout

from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn import metrics
```

Загрузка данных и выбор "предпочитаемой" страны

In [ ]:
```
world_cities = pd.read_csv("worldcities.csv", header=None, skiprows=1)
world_cities = world_cities.to_numpy()

target_country = 'Russia'
world_cities[:,3] = np.where(world_cities[:,2] == target_country, 1, 0)

print(world_cities)

X = world_cities[:,0:2]
y = world_cities[:,3]

X = X.astype(np.float32)
y = y.astype(np.float32)
```

```
[[35.685 139.7514 'Japan' 0]
 [40.6943 -73.9249 'United States' 0]
 [19.4424 -99.131 'Mexico' 0]
 ...
 [69.651 162.3336 'Russia' 1]
 [74.0165 111.51 'Russia' 1]
 [61.1333 -100.8833 'Canada' 0]]
```

1. Обучение

Структура нейронной сети:

- Входной слой 2
- Внутренний слой 128
- Активационный слой ReLU

- Слой нормализации (по batch)
- Слой DropOut (0.3)
- Внутренний слой 64
- Активационный слой ReLU
- Слой DropOut (0.3)
- Внутренний слой 1
- Активационный слой Sigmoid

Также важно отметить, что модель обучается на 80% данных (тех.задание) и для обучения используется предподготовка данных - происходит нормалиация данных, а так же во время обучения влияние верных значений (с меткой 1) имеет больший вес (10 кратный), по сравнению с неверным значением (с меткой 0), это сделано изходя из того, что данные имеют явный дисбаланс.

Нормализуем данные

In [ ]:
```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2)
```

Структура модели

In [ ]:
```
model = Sequential()
model.add(Dense(128, input_dim=2, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.3))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer=tf.optimizers.Adam(learning_rate=0.001), 

class_weights = {0: 1, 1: 10}
```

Обучение и проверка

In [ ]:
```
history = model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=1, class_weight=class_

y_pred = (model.predict(X_test) > 0.5).astype(int)
```

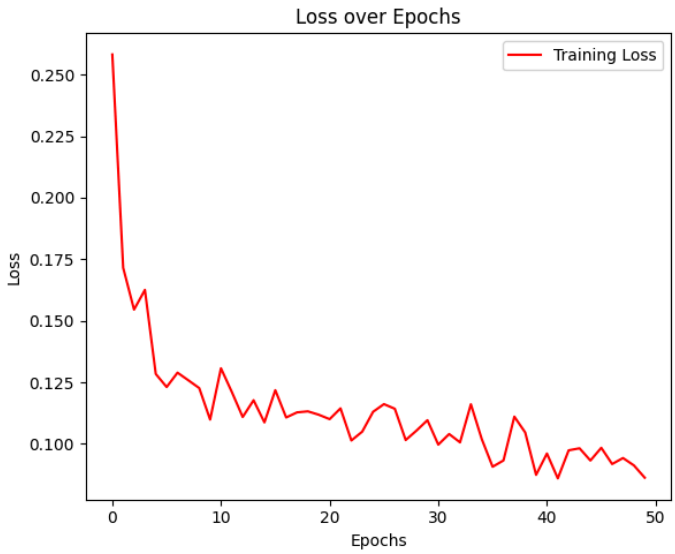
```
Epoch 1/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9049 - loss: 0.3798
Epoch 2/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9592 - loss: 0.1888
Epoch 3/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9646 - loss: 0.1650
Epoch 4/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9705 - loss: 0.1369
Epoch 5/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9654 - loss: 0.1328
Epoch 6/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9735 - loss: 0.1070
Epoch 7/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9661 - loss: 0.1361
Epoch 8/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9727 - loss: 0.1294
Epoch 9/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9708 - loss: 0.1127
Epoch 10/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9737 - loss: 0.1010
Epoch 11/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9678 - loss: 0.1258
Epoch 12/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9699 - loss: 0.1098
Epoch 13/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9730 - loss: 0.1071
Epoch 14/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9692 - loss: 0.1144
Epoch 15/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9715 - loss: 0.1113
Epoch 16/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9743 - loss: 0.1147
Epoch 17/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9695 - loss: 0.1144
Epoch 18/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9737 - loss: 0.1083
Epoch 19/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9745 - loss: 0.1081
Epoch 20/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9691 - loss: 0.1087
Epoch 21/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9721 - loss: 0.1055
Epoch 22/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9696 - loss: 0.1178
Epoch 23/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9737 - loss: 0.0947
Epoch 24/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9735 - loss: 0.0982
Epoch 25/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9693 - loss: 0.1244
Epoch 26/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9668 - loss: 0.1170
Epoch 27/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9661 - loss: 0.1268
Epoch 28/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9723 - loss: 0.1120
Epoch 29/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9755 - loss: 0.1049
Epoch 30/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9701 - loss: 0.1176
Epoch 31/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9742 - loss: 0.1007
Epoch 32/50
```

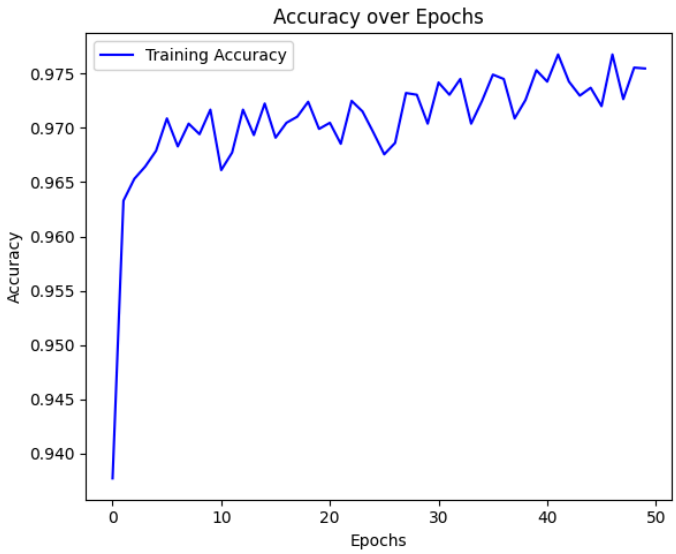
```
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9752 - loss: 0.1030
Epoch 33/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9763 - loss: 0.0937
Epoch 34/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9718 - loss: 0.1123
Epoch 35/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9736 - loss: 0.0981
Epoch 36/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9761 - loss: 0.0828
Epoch 37/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9723 - loss: 0.0984
Epoch 38/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9660 - loss: 0.1045
Epoch 39/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9704 - loss: 0.1012
Epoch 40/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9751 - loss: 0.0845
Epoch 41/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9764 - loss: 0.0892
Epoch 42/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9764 - loss: 0.0832
Epoch 43/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9752 - loss: 0.0935
Epoch 44/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9741 - loss: 0.0942
Epoch 45/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9733 - loss: 0.0891
Epoch 46/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9673 - loss: 0.1122
Epoch 47/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9759 - loss: 0.0894
Epoch 48/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9720 - loss: 0.0933
Epoch 49/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9784 - loss: 0.0754
Epoch 50/50
388/388 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9751 - loss: 0.0786
97/97 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step
```

Графики потерь и точности

(чтобы убедиться что мы не переобучились/недообучились)

In [ ]:
```python
fig, axs = plt.subplots(1, 2, figsize=(12, 5))

axs[0].plot(history.history['loss'], label='Training Loss', color='red')
axs[0].set_title('Loss over Epochs')
axs[0].set_xlabel('Epochs')
axs[0].set_ylabel('Loss')
axs[0].legend()

axs[1].plot(history.history['accuracy'], label='Training Accuracy', color='blue')
axs[1].set_title('Accuracy over Epochs')
axs[1].set_xlabel('Epochs')
axs[1].set_ylabel('Accuracy')
axs[1].legend()

plt.tight_layout()
plt.show()
```





Расчёт метрики p

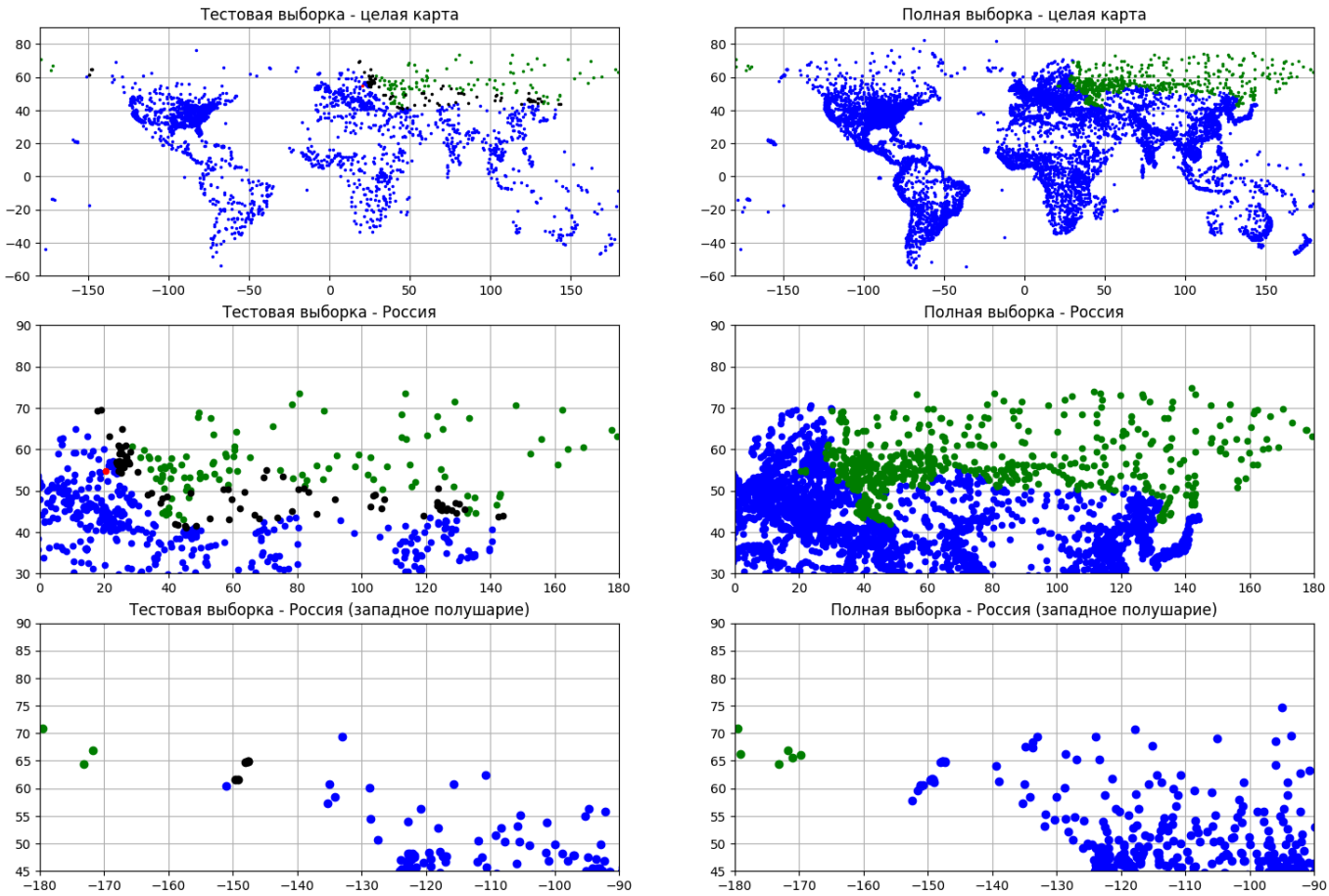
In [ ]:
```
TN_d = np.empty((0,2),dtype=float)
TP_d = np.empty((0,2),dtype=float)
FN_d = np.empty((0,2),dtype=float)
FP_d = np.empty((0,2),dtype=float)

y_pred = y_pred.T[0]
X_test = scaler.inverse_transform(X_test)

TP_d = X_test[(y_test == 1) & (y_pred == 1)]
TN_d = X_test[(y_test == 0) & (y_pred == 0)]
FP_d = X_test[(y_test == 0) & (y_pred == 1)]
FN_d = X_test[(y_test == 1) & (y_pred == 0)]

TP = len(TP_d)
TN = len(TN_d)
FP = len(FP_d)
FN = len(FN_d)

print(TN,TP,FN,FP)
print(TN+TP+FN+FP)

p = 0.5 * ((TP / (TP + FN)) + (TN / (TN + FP)))
print(f'Metric p: {p}')
```

```
2898 111 1 89
3099
Metric p: 0.9806378234253192
```

Рассмотрение итогового результата

In [ ]:
```
X_0 = X[y == 0]
X_1 = X[y == 1]

fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(18, 12))

axes[0, 0].plot(TN_d[:,1], TN_d[:,0], "b.", label='TN', ms=3)
axes[0, 0].plot(TP_d[:,1], TP_d[:,0], "g.", label='TP', ms=3)
axes[0, 0].plot(FN_d[:,1], FN_d[:,0], "r.", label='FN', ms=3)
axes[0, 0].plot(FP_d[:,1], FP_d[:,0], "k.", label='FP', ms=3)
axes[0, 0].set_title('Тестовая выборка - целая карта')
axes[0, 0].grid(True)
axes[0, 0].set_xlim(-180, 180)
```

```
axes[0, 0].set_ylim(-60, 90)

axes[0, 1].plot(X_0[:,1], X_0[:,0], "b.", label='T', ms=3)
axes[0, 1].plot(X_1[:,1], X_1[:,0], "g.", label='F', ms=3)
axes[0, 1].set_title('Полная выборка - целая карта')
axes[0, 1].grid(True)
axes[0, 1].set_xlim(-180, 180)
axes[0, 1].set_ylim(-60, 90)

axes[1, 0].plot(TN_d[:,1], TN_d[:,0], "b.", label='TN', ms=9)
axes[1, 0].plot(TP_d[:,1], TP_d[:,0], "g.", label='TP', ms=9)
axes[1, 0].plot(FN_d[:,1], FN_d[:,0], "r.", label='FN', ms=9)
axes[1, 0].plot(FP_d[:,1], FP_d[:,0], "k.", label='FP', ms=9)
axes[1, 0].set_title('Тестовая выборка - Россия')
axes[1, 0].grid(True)
axes[1, 0].set_xlim(0, 180)
axes[1, 0].set_ylim(30, 90)

axes[1, 1].plot(X_0[:,1], X_0[:,0], "b.", label='T', ms=9)
axes[1, 1].plot(X_1[:,1], X_1[:,0], "g.", label='F', ms=9)
axes[1, 1].set_title('Полная выборка - Россия')
axes[1, 1].grid(True)
axes[1, 1].set_xlim(0, 180)
axes[1, 1].set_ylim(30, 90)

axes[2, 0].plot(TN_d[:,1], TN_d[:,0], "b.", label='TN', ms=12)
axes[2, 0].plot(TP_d[:,1], TP_d[:,0], "g.", label='TP', ms=12)
axes[2, 0].plot(FN_d[:,1], FN_d[:,0], "r.", label='FN', ms=12)
axes[2, 0].plot(FP_d[:,1], FP_d[:,0], "k.", label='FP', ms=12)
axes[2, 0].set_title('Тестовая выборка - Россия (западное полушарие)')
axes[2, 0].grid(True)
axes[2, 0].set_xlim(-180, -90)
axes[2, 0].set_ylim(45, 90)

axes[2, 1].plot(X_0[:,1], X_0[:,0], "b.", label='T', ms=12)
axes[2, 1].plot(X_1[:,1], X_1[:,0], "g.", label='F', ms=12)
# axes[2, 1].set_xlim(0, 6)
# axes[2, 1].set_ylim(0, 40)
axes[2, 1].set_title('Полная выборка - Россия (западное полушарие)')
axes[2, 1].grid(True)
axes[2, 1].set_xlim(-180, -90)
axes[2, 1].set_ylim(45, 90)

plt.show()
```



Итог

Нейронная сеть с данной структурой способна обучиться за 50 эпох и выдавать стабильный результат $p = 0.98\text{-}0.99$.

2. Расчёт размера

Напомню структуру нейронной сети:

- Входной слой 2
- Внутренний слой 128
- Активационный слой ReLU
- Слой нормализации (по batch)
- Слой DropOut (0.3)
- Внутренний слой 64
- Активационный слой ReLU
- Слой DropOut (0.3)
- Внутренний слой 1
- Активационный слой Sigmoid
- Оптимизатор Adam
- Скорость обучения 0.001

1. **ReLU**

Не имеет операций умножения.

2. **Полносвязные слои**

Каждый такой слой содержит в себе матрицу умножения, как например с 2 на 128 нейронов будет:

$$2 * 128 = 256 \text{ умножений}$$

3. **Слои DropOut**

Они выключают часть нейроном с предшествующего слоя, как например с слоя 64 через DropOut(0.3) на слой с 1 нейроном

$$64 * (1 - 0.3) * 1 \approx 45 \text{ умножений}$$

4. **BatchNormalization**

Данный слой довольно сложный, для расчёта числа умножений надо рассмотреть его структуру. Сперва вычисляется среднее

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x_i \text{ (1 умножение на признак)}$$

Затем дисперсию:

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu)^2 \text{ } (m + 1 \text{ умножений на признак})$$

Нормализация:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \text{ где } \epsilon - \text{маленькое число (1 умножение на признак)}$$

Масштабирование и сдвиг:

$$\hat{y}_i = \gamma \hat{x}_i + \beta, \text{ где } \gamma, \beta - \text{параметры (1 умножение на признак)}$$

Получается в таком слое происходит сумарно $m + 4$ умножений на признак, где m - размер батча.

На самом деле в реализациях чаще всего используют не просто рассчитанные значения μ и σ^2, а их скользящее среднее (что ещё больше увеличит число умножений)

5. **Оптимизатор Adam**

Внутри оптимизатора, на каждом слое происходит:

- Вычисление первого момента

$$m_{t+1} = \beta_1 * m_t + (1 - \beta_1) * g_t$$

- Вычисление второго момента

$$v_{t+1} = \beta_2 * v_t + (1 - \beta_2) * g_t^2$$

- Коррекция смещения

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- Обновление параметров

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}} + \epsilon}$$

С учётом того, что для каждого слоя надо ещё предварительно посчитать g_t

Подсчёт

- $2 * 128 = 256$ - переход с 2 нейронов на 128
- $128 * (32 + 4) = 4608$ - нормализация по батчу
- $128 * (1 - 0.3) * 64 = 5734.4$ - переход с 128 нейронов на 64 с выключением 30%
- $64 * (1 - 0.3) * 1 = 44.8$ - переход с 64 нейронов на 1 с выключением 30%
- $256 + 5734.4 + 44.8 = 6035.2$ - расчёт градиента ошибки для слоёв нейронов
- $128 * 7 = 896$ - расчёт градиента ошибки для нормализации по батчу ((1 на $\hat{x}$, 1 на μ, 2 на σ, 3 на x) $\times$ признаков)
- $6035.2 * 9 = 54316.8$ - применение оптимизатора Adam ((2 на 1-й момент, 3 на 2-1 момент, 2 на коррекцию, 2 на обновление веса) $\times$ градиент ошибки)

На одну эпоху, с учётом того, что размер батча 32 будет

$$(256 + 5734.4 + 44.8 + 6035.2 + 896 + 54316.8) * 32 + 4608 = 2157670.4$$

Итого 107883520 **умножений**

3. Оптимизация

Попробуем теперь максимально снизить число умножений в нейронной сети.

Процесс поиска представлял обрезание и обнищание нейронной сети со структурой выше: сначала это были объективные вещи, такие как: отказ от нормализации по батчу и использование простого оптимизатора SGD, а не Adam. Далее было решено отбросить и регуляризацию, так как большого эффекта от неё не будет на малом количестве эпох.

В остатке, происходило уменьшение числа нейронов и даже слоёв пока ещё средний результат метрики p оставался в среднем выше 0.97 и в итоге была получена такая структура нейронной сети

Структура

- 2 входных нейрона
- 16 внутренних, активация ReLU
- 1 выходной, активация Sigmoid

Обучение с дисбалансом классов в 20 раз, количество эпох 20, оптимизатор SGD, скорость обучения 0.4

Так как нейронная сеть не всегда стабильно и чётко отрабатывает и результат очень сильно зависит от разбиения начальных данных и выборке примеров для обучения (всего обучение может быть на $15474 * 0.8 \approx 11700$ примерах, а в данном случае будет задействовано $32 * 20 = 640$ случайно выбранных). Поэтому для справедливой оценки качества предсказаний данной нейронной сети, будем усреднять запуски обучения, пока не придём к определённому пределу (5 запусков без изменений среднего значения метрики p)

In [ ]:

```
import warnings
warnings.filterwarnings('ignore', category=UserWarning)

df = pd.read_csv('worldcities.csv')
target_country = "Russia"
df['new_target'] = (df['country'] == target_country).astype(int)

X = df[['lat', 'lng']].values
y = df['new_target'].values

layer_1 = 16
true_class_weight = 20

num_epochs = 20

learning_rate = 0.4
model_optimizer = tf.optimizers.SGD(learning_rate=learning_rate)

num_runs = 0

losses = []
accuracies = []

TN_d_x = []
TN_d_y = []
TP_d_x = []
TP_d_y = []
FN_d_x = []
FN_d_y = []
FP_d_x = []
FP_d_y = []

pp = []
prev_rounded_mean_p = 0
while (num_runs < 5):
    # делим данные
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)
        X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2)

        # строим модель
        model = Sequential()
        model.add(Dense(layer_1, input_dim=2, activation='relu'))
        model.add(Dense(1, activation='sigmoid'))

        model_optimizer = tf.optimizers.SGD(learning_rate=learning_rate)
        model.compile(loss='binary_crossentropy', optimizer=model_optimizer, metrics=['accura

        # обучаем
        class_weights = {0: 1, 1: true_class_weight}
        history = model.fit(X_train, y_train, epochs=num_epochs, batch_size=32, verbose=0, cl

        # тестируем
        y_pred = (model.predict(X_test) > 0.5).astype(int)

        # запоминаем точность и потери
        losses.append(history.history['loss'])
        accuracies.append(history.history['accuracy'])

        TN_d = np.empty((0,2),dtype=float)
        TP_d = np.empty((0,2),dtype=float)
        FN_d = np.empty((0,2),dtype=float)
        FP_d = np.empty((0,2),dtype=float)
```

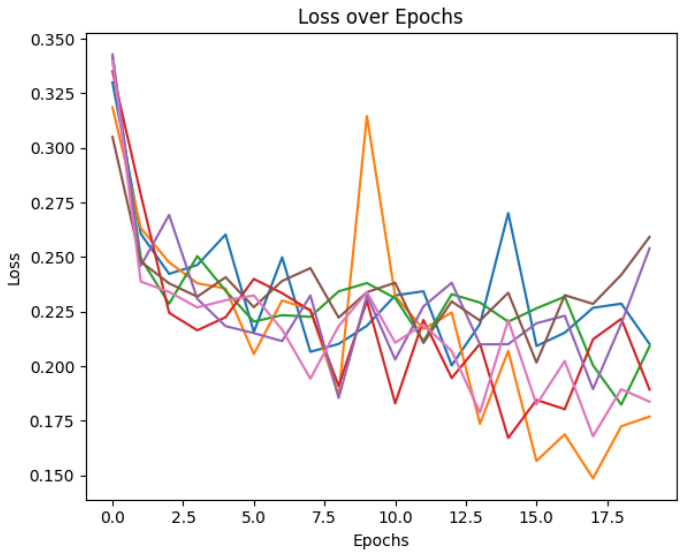
```
        y_pred = y_pred.T[0]
        X_test = scaler.inverse_transform(X_test)

        TP_d = X_test[(y_test == 1) & (y_pred == 1)]
        TN_d = X_test[(y_test == 0) & (y_pred == 0)]
        FP_d = X_test[(y_test == 0) & (y_pred == 1)]
        FN_d = X_test[(y_test == 1) & (y_pred == 0)]

        TN_d_x.append(TN_d[:,1])
        TN_d_y.append(TN_d[:,0])
        TP_d_x.append(TP_d[:,1])
        TP_d_y.append(TP_d[:,0])
        FP_d_x.append(FP_d[:,1])
        FP_d_y.append(FP_d[:,0])
        FN_d_x.append(FN_d[:,1])
        FN_d_y.append(FN_d[:,0])

        TP = len(TP_d)
        TN = len(TN_d)
        FP = len(FP_d)
        FN = len(FN_d)

        print(TN,TP,FN,FP)
        print(TN+TP+FN+FP)

        p = 0.5 * ((TP / (TP + FN)) + (TN / (TN + FP)))
        pp.append(p)
        mean_p = sum(pp) / len(pp)
        print(f'Metric p: {p}')
        print("Prev pp:", prev_rounded_mean_p, "now", mean_p)
        if (prev_rounded_mean_p == round(mean_p, 2)):
                num_runs += 1
        else:
                num_runs = 0
        prev_rounded_mean_p = round(mean_p, 2)
```

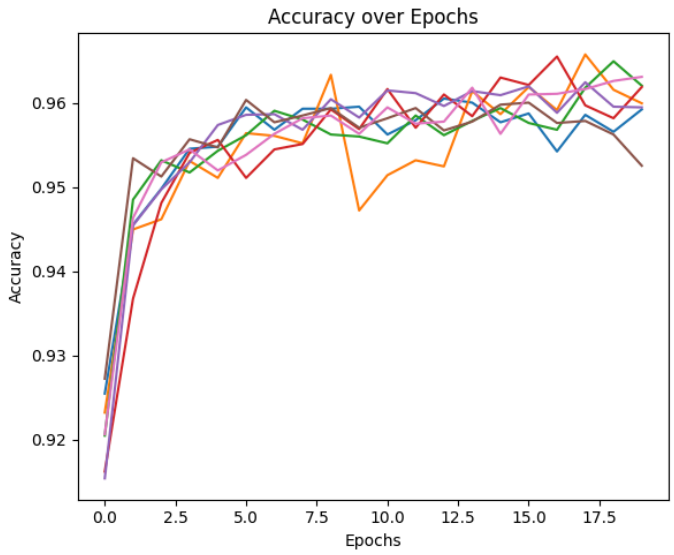
```
97/97 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
2774 113 0 212
3099
Metric p: 0.9645010046885465
Prev pp: 0 now 0.9645010046885465
97/97 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
2896 117 1 85
3099
Metric p: 0.9815057511129811
Prev pp: 0.96 now 0.9730033779007639
97/97 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step
2875 107 3 114
3099
Metric p: 0.9672937133124486
Prev pp: 0.97 now 0.9711001563713255
97/97 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
2909 114 2 74
3099
Metric p: 0.9789756898285688
Prev pp: 0.97 now 0.9730690397356363
97/97 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
2793 114 0 192
3099
Metric p: 0.9678391959798995
Prev pp: 0.97 now 0.9720230709844889
97/97 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
2840 110 4 145
3099
Metric p: 0.9581680331481972
Prev pp: 0.97 now 0.9697138980117735
97/97 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
2892 118 1 88
3099
Metric p: 0.9810332186565902
Prev pp: 0.97 now 0.9713309438181759
```

In [ ]:

```python
mean_p = sum(pp) / len(pp)
print(f'Mean Metric p: {mean_p}')

fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# График потерь (loss)
for i in range(len(pp)):
        axs[0].plot(losses[i])
axs[0].set_title('Loss over Epochs')
axs[0].set_xlabel('Epochs')
axs[0].set_ylabel('Loss')
# axs[0].legend()

# График точности (accuracy)
for i in range(len(pp)):
        axs[1].plot(accuracies[i])
axs[1].set_title('Accuracy over Epochs')
axs[1].set_xlabel('Epochs')
axs[1].set_ylabel('Accuracy')
# axs[1].legend()

# Показать графики
plt.tight_layout()
plt.show()
```

```
Mean Metric p: 0.9713309438181759
```

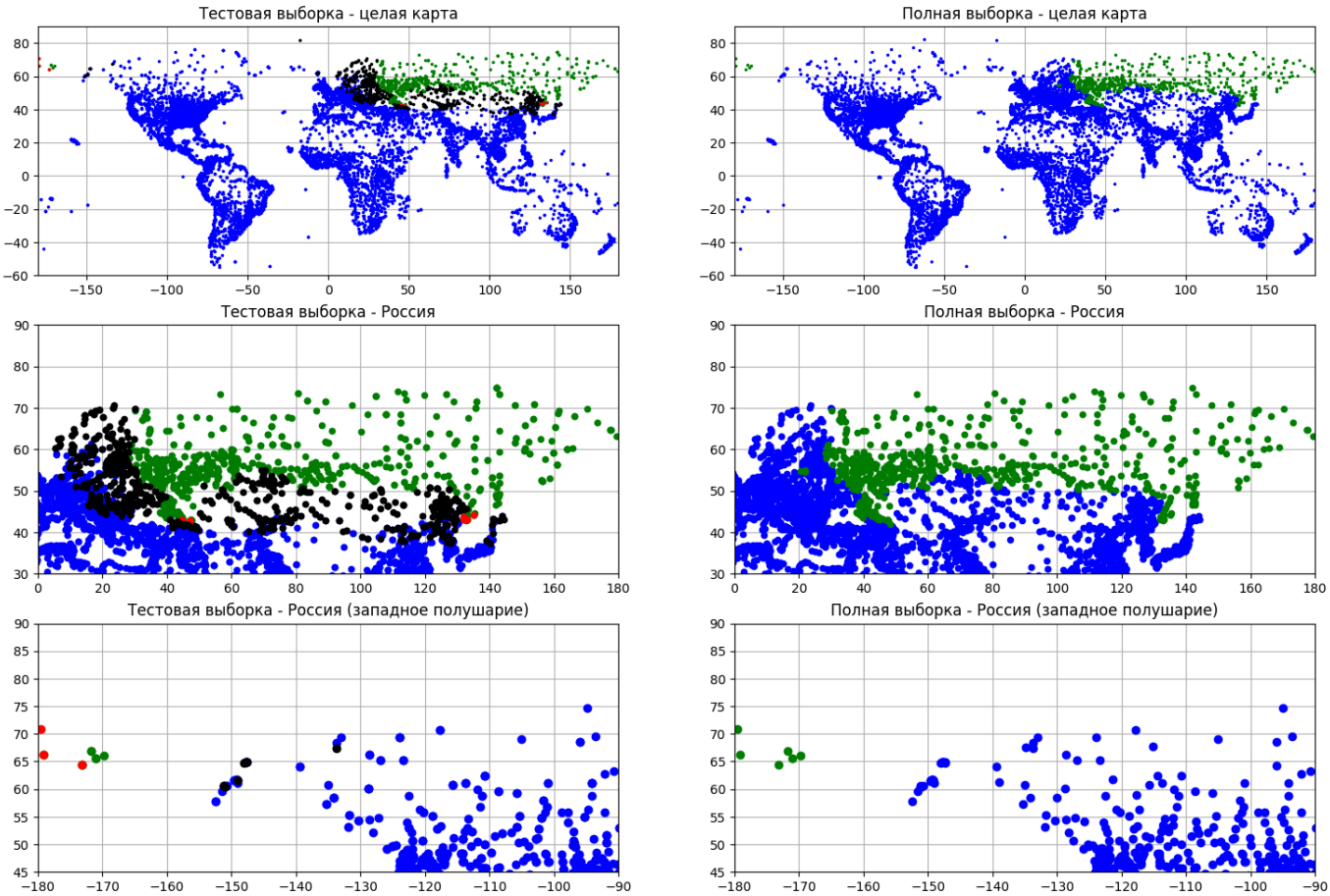




In [ ]:

```
X_0 = X[y == 0]
X_1 = X[y == 1]

TN_d_x = np.concatenate(TN_d_x)
TN_d_y = np.concatenate(TN_d_y)
TP_d_x = np.concatenate(TP_d_x)
TP_d_y = np.concatenate(TP_d_y)
FN_d_x = np.concatenate(FN_d_x)
FN_d_y = np.concatenate(FN_d_y)
FP_d_x = np.concatenate(FP_d_x)
FP_d_y = np.concatenate(FP_d_y)

fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(18, 12))

axes[0, 0].plot(TN_d_x, TN_d_y, "b.", label='TN', ms=3)
axes[0, 0].plot(TP_d_x, TP_d_y, "g.", label='TP', ms=3)
axes[0, 0].plot(FN_d_x, FN_d_y, "r.", label='FN', ms=3)
axes[0, 0].plot(FP_d_x, FP_d_y, "k.", label='FP', ms=3)
axes[0, 0].set_title('Тестовая выборка - целая карта')
axes[0, 0].grid(True)
axes[0, 0].set_xlim(-180, 180)
axes[0, 0].set_ylim(-60, 90)

axes[0, 1].plot(X_0[:,1], X_0[:,0], "b.", label='T', ms=3)
axes[0, 1].plot(X_1[:,1], X_1[:,0], "g.", label='F', ms=3)
axes[0, 1].set_title('Полная выборка - целая карта')
axes[0, 1].grid(True)
axes[0, 1].set_xlim(-180, 180)
axes[0, 1].set_ylim(-60, 90)

axes[1, 0].plot(TN_d_x, TN_d_y, "b.", label='TN', ms=9)
axes[1, 0].plot(TP_d_x, TP_d_y, "g.", label='TP', ms=9)
axes[1, 0].plot(FN_d_x, FN_d_y, "r.", label='FN', ms=9)
axes[1, 0].plot(FP_d_x, FP_d_y, "k.", label='FP', ms=9)
axes[1, 0].set_title('Тестовая выборка - Россия')
axes[1, 0].grid(True)
axes[1, 0].set_xlim(0, 180)
axes[1, 0].set_ylim(30, 90)

axes[1, 1].plot(X_0[:,1], X_0[:,0], "b.", label='T', ms=9)
axes[1, 1].plot(X_1[:,1], X_1[:,0], "g.", label='F', ms=9)
axes[1, 1].set_title('Полная выборка - Россия')
axes[1, 1].grid(True)
axes[1, 1].set_xlim(0, 180)
axes[1, 1].set_ylim(30, 90)
```

```
axes[2, 0].plot(TN_d_x, TN_d_y, "b.", label='TN', ms=12)
axes[2, 0].plot(TP_d_x, TP_d_y, "g.", label='TP', ms=12)
axes[2, 0].plot(FN_d_x, FN_d_y, "r.", label='FN', ms=12)
axes[2, 0].plot(FP_d_x, FP_d_y, "k.", label='FP', ms=12)
axes[2, 0].set_title('Тестовая выборка - Россия (западное полушарие)')
axes[2, 0].grid(True)
axes[2, 0].set_xlim(-180, -90)
axes[2, 0].set_ylim(45, 90)

axes[2, 1].plot(X_0[:,1], X_0[:,0], "b.", label='T', ms=12)
axes[2, 1].plot(X_1[:,1], X_1[:,0], "g.", label='F', ms=12)
# axes[2, 1].set_xlim(0, 6)
# axes[2, 1].set_ylim(0, 40)
axes[2, 1].set_title('Полная выборка - Россия (западное полушарие)')
axes[2, 1].grid(True)
axes[2, 1].set_xlim(-180, -90)
axes[2, 1].set_ylim(45, 90)

plt.show()
```



Подсчёт

- $2 * 16 = 32$ - переход с 2 нейронов на 16
- $16 * 1 = 16$ - переход с 16 нейронов на 1
- $32 + 16 = 48$ - расчёт градиента ошибки для слоёв нейронов
- 48 - применение оптимизатора SGD (каждый градиент на скорость обучения)

На одну эпоху, с учётом того, что размер батча 32 будет

$$32 * (32 + 16 + 48 + 48) = 4608$$

Итого 92160 **умножений** (результат улучшился в 1000 раз!)

Такой результат был достигнут из-за (1) оптимизаций, (2) небольшого снижения требований, что на самом деле тоже имеет весьма большое значение, так как достигнуть результата $p = 0.97$ в десятки раз проще чем $p = 0.98$