

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ  
УНИВЕРСИТЕТ»

ФАКУЛЬТЕТ ЭЛЕКТРОННО-ИНФОРМАЦИОННЫХ СИСТЕМ

Кафедра интеллектуальных информационных технологий

Отчет по лабораторной работе №4

Специальность ИИ(з)

Выполнил  
А. Ю. Кураш,  
студент группы ИИ-24

Проверил  
Андренко К.В.,  
Преподаватель-стажер кафедры ИИТ,  
«\_\_» k \_\_\_\_\_ 2025 г.

Брест 2025

**Цель:** научиться осуществлять предобучение нейронных сетей с помощью RBM

### **Общее задание**

1. Взять за основу нейронную сеть из лабораторной работы №3. Выполнить обучение с предобучением, используя стек ограниченных машин Больцмана (RBM – Restricted Boltzmann Machine), алгоритм которого изложен в лекции. Условие останова (например, по количеству эпох) при обучении отдельных слоев как RBM выбрать самостоятельно.
2. Сравнить результаты, полученные при
  - обучении без предобучения (ЛР 3);
  - обучении с предобучением, используя автоэнкодерный подход (ЛР3);
  - обучении с предобучением, используя RBM.
3. Обучить модели на данных из ЛР 2, сравнить результаты по схеме из пункта 2;
4. Сделать выводы, оформить отчет по выполненной работе, загрузить исходный код и отчет в соответствующий репозиторий на github.

### **Задание по вариантам**

8	<a href="https://archive.ics.uci.edu/dataset/162/forest+fires">https://archive.ics.uci.edu/dataset/162/forest+fires</a>	регрессия	area
---	---	-----------	------

#### **Код программы:**

```
import os
import math
import random
import numpy as np
import pandas as pd
import json
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import TensorDataset, DataLoader

# --- Фиксация seed для воспроизводимости
seed = 42
```

```

random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

# --- Параметры
# Датасеты
DATA_URL_FIRES = "https://archive.ics.uci.edu/ml/machine-learning-databases/forest-
fires/forestfires.csv"
DATA_URL_RICE = "https://download.mlcc.google.com/mledu-
datasets/Rice_Cammeo_Osmancik.csv"

# Общие параметры
BATCH_SIZE = 32
WEIGHT_DECAY = 1e-5
HIDDEN_SIZES = [64, 32, 16, 8] # Архитектура (4 скрытых слоя)

# Параметры для Forest Fires (Регрессия)
EPOCHS_MLP_FIRES = 200
EPOCHS_AE_FIRES = 120
EPOCHS_RBM_FIRES = 50 # (Выбрано самостоятельно, как в задании)
EPOCHS_FINE_TUNE_FIRES = 150
LR_MLP_FIRES = 1e-3
LR_PRETRAIN_FIRES = 1e-3
LR_FINE_FIRES = 5e-4

# Параметры для Rice (Классификация)
EPOCHS_MLP_RICE = 100
EPOCHS_AE_RICE = 80
EPOCHS_RBM_RICE = 40 # (Выбрано самостоятельно)
EPOCHS_FINE_TUNE_RICE = 100
LR_MLP_RICE = 1e-3
LR_PRETRAIN_RICE = 1e-3
LR_FINE_RICE = 5e-4

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# --- Загрузка
def load_data(path_or_url):
    if os.path.exists(path_or_url):
        df = pd.read_csv(path_or_url)
    else:
        print(f"Downloading from {path_or_url}...")
        df = pd.read_csv(path_or_url)
    return df

```

```

# --- Вспомогательная функция для JSON
def safe_json_dump(data):
    return json.dumps(data, indent=2, default=float)

#
=====
=====
# --- ЧАСТЬ 1: ЗАДАЧА РЕГРЕССИИ (FOREST FIRES)
#
=====
=====
print("\n--- ЧАСТЬ 1: ЗАДАЧА РЕГРЕССИИ (FOREST FIRES) ---")

# --- Предобработка (Fires)
df_fires = load_data(DATA_URL_FIRES)

df_fires['month'] = df_fires['month'].astype(str)
df_fires['day'] = df_fires['day'].astype(str)
cat = df_fires[['month', 'day']]
enc = OneHotEncoder(sparse_output=False, drop='first')
cat_enc = enc.fit_transform(cat)

num = df_fires[['X', 'Y', 'FFMC', 'DMC', 'DC', 'ISI', 'temp', 'RH', 'wind', 'rain']].values.astype(float)
X_raw_fires = np.hstack([num, cat_enc])
y_raw_fires = df_fires['area'].values.astype(float)
y_log_fires = np.log1p(y_raw_fires) # Логарифмируем цель

scaler_fires = StandardScaler()
X_fires = scaler_fires.fit_transform(X_raw_fires)

X_train_f, X_test_f, y_train_f, y_test_f, ylog_train_f, ylog_test_f = train_test_split(
    X_fires, y_raw_fires, y_log_fires, test_size=0.2, random_state=seed
)

X_train_f_t = torch.tensor(X_train_f, dtype=torch.float32)
X_test_f_t = torch.tensor(X_test_f, dtype=torch.float32)
ytrain_f_t = torch.tensor(ylog_train_f, dtype=torch.float32).unsqueeze(1)
ytest_f_t = torch.tensor(ylog_test_f, dtype=torch.float32).unsqueeze(1)

train_ds_f = TensorDataset(X_train_f_t, ytrain_f_t)
test_ds_f = TensorDataset(X_test_f_t, ytest_f_t)
train_loader_f = DataLoader(train_ds_f, batch_size=BATCH_SIZE, shuffle=True)
test_loader_f = DataLoader(test_ds_f, batch_size=BATCH_SIZE, shuffle=False)

input_dim_fires = X_fires.shape[1]

# --- Вспомогательные функции (Регрессия)
def train_model_regression(model, train_loader, epochs=100, lr=1e-3, weight_decay=0.0):
    model = model.to(device)

```

```

opt = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=weight_decay)
criterion = nn.MSELoss()
for epoch in range(epochs):
    model.train()
    for xb, yb in train_loader:
        xb, yb = xb.to(device), yb.to(device)
        opt.zero_grad()
        out = model(xb)
        loss = criterion(out, yb)
        loss.backward()
        opt.step()
    return model

def evaluate_model_regression(model, X_t, y_raw_true):
    model.eval()
    with torch.no_grad():
        pred_log = model(X_t.to(device)).cpu().numpy().reshape(-1)

    # Обратное преобразование (из логарифма)
    pred_area = np.expm1(pred_log)
    # Ограничиваем отрицательные предсказания нулем
    pred_area[pred_area < 0] = 0

    true_area = y_raw_true.reshape(-1)
    eps = 1e-6
    rmse = np.sqrt(np.mean((pred_area - true_area)**2))
    mae = np.mean(np.abs(pred_area - true_area))

    nonzero_mask = true_area > 0
    if nonzero_mask.sum() > 0:
        mape = np.mean(np.abs((pred_area[nonzero_mask] - true_area[nonzero_mask]) /
        (true_area[nonzero_mask] + eps))) * 100
    else:
        mape = float('nan')

    return {'RMSE': rmse, 'MAE': mae, 'MAPE_%_on_nonzero_targets': mape}

# --- Модель MLP (Общая)
class MLP(nn.Module):
    def __init__(self, input_dim, hidden_sizes, output_dim=1):
        super().__init__()
        layers = []
        prev = input_dim
        for h in hidden_sizes:
            layers.append(nn.Linear(prev, h))
            layers.append(nn.ReLU())
            prev = h
        layers.append(nn.Linear(prev, output_dim)) # Выходной слой
        self.net = nn.Sequential(*layers)
    def forward(self, x):

```

```

return self.net(x)

# 1.1 MLP (без предобучения)
print("Training MLP from scratch (Fires)...")
torch.manual_seed(seed)
model_plain_f = MLP(input_dim=input_dim_fires, hidden_sizes=HIDDEN_SIZES,
output_dim=1)
model_plain_f = train_model_regression(
    model_plain_f, train_loader_f, epochs=EPOCHS_MLP_FIRES, lr=LR_MLP_FIRES,
weight_decay=WEIGHT_DECAY
)
metrics_plain_f_train = evaluate_model_regression(model_plain_f, X_train_f_t, y_train_f)
metrics_plain_f_test = evaluate_model_regression(model_plain_f, X_test_f_t, y_test_f)

# --- 1.2 Autoencoders (предобучение)
class Autoencoder(nn.Module):
    def __init__(self, in_dim, hidden_dim):
        super().__init__()
        self.encoder = nn.Sequential(nn.Linear(in_dim, hidden_dim), nn.ReLU())
        self.decoder = nn.Sequential(nn.Linear(hidden_dim, in_dim))
    def forward(self, x):
        z = self.encoder(x)
        out = self.decoder(z)
        return out
    def encode(self, x):
        return self.encoder(x)

def train_autoencoder(ae, data_tensor, epochs=100, lr=1e-3):
    ae = ae.to(device)
    opt = torch.optim.Adam(ae.parameters(), lr=lr)
    criterion = nn.MSELoss()
    ds = TensorDataset(data_tensor, data_tensor)
    loader = DataLoader(ds, batch_size=BATCH_SIZE, shuffle=True)
    for epoch in range(epochs):
        ae.train()
        for xb, _ in loader:
            xb = xb.to(device)
            opt.zero_grad()
            out = ae(xb)
            loss = criterion(out, xb) # Восстанавливаем сам X
            loss.backward()
            opt.step()
    return ae

print("Pretraining Autoencoders (Fires)...")
X_train_tensor_f = X_train_f_t.clone().to(device)
encoders_f = []
current_input_f = X_train_tensor_f
for h in HIDDEN_SIZES:
    in_dim = current_input_f.shape[1]

```

```

ae = Autoencoder(in_dim, h)
ae = train_autoencoder(ae, current_input_f, epochs=EPOCHS_AE_FIRES,
lr=LR_PRETRAIN_FIRES)
with torch.no_grad():
    encoded = ae.encode(current_input_f.to(device)).cpu()
encoders_f.append(ae.encoder)
current_input_f = torch.tensor(encoded, dtype=torch.float32)

```

# Сборка AE-MLP

```

class MLP_Pretrained_AE(nn.Module):
    def __init__(self, input_dim, encoders, output_dim=1):
        super().__init__()
        layers = []
        prev = input_dim
        for enc in encoders:
            # enc[0] - это Linear слой из nn.Sequential
            lin_layer = nn.Linear(prev, enc[0].out_features)
            # Копируем веса
            lin_layer.weight.data = enc[0].weight.data.clone()
            lin_layer.bias.data = enc[0].bias.data.clone()
            layers.append(lin_layer)
            layers.append(nn.ReLU())
            prev = enc[0].out_features
        layers.append(nn.Linear(prev, output_dim)) # Выходной слой
        self.net = nn.Sequential(*layers)
    def forward(self, x):
        return self.net(x)

```

```

print("Fine-tuning AE-MLP (Fires)...")

```

```

torch.manual_seed(seed)

```

```

model_pre_ae_f = MLP_Pretrained_AE(input_dim=input_dim_fires, encoders=encoders_f,
output_dim=1).to(device)

```

```

model_pre_ae_f = train_model_regression(

```

```

    model_pre_ae_f, train_loader_f, epochs=EPOCHS_FINE_TUNE_FIRES,
lr=LR_FINE_FIRES, weight_decay=WEIGHT_DECAY

```

```

)
metrics_pre_ae_f_train = evaluate_model_regression(model_pre_ae_f, X_train_f_t, y_train_f)
metrics_pre_ae_f_test = evaluate_model_regression(model_pre_ae_f, X_test_f_t, y_test_f)

```

# --- 1.3 RBM (предобучение) ---

# Гауссовско-Бернуллиевская RBM (т.к. входные данные непрерывные, стандартизированные)

```

class RBM(nn.Module):
    def __init__(self, n_visible, n_hidden):
        super(RBM, self).__init__()
        self.W = nn.Parameter(torch.randn(n_hidden, n_visible) * 1e-2)
        self.hb = nn.Parameter(torch.zeros(n_hidden))
        self.vb = nn.Parameter(torch.zeros(n_visible))

    def sample_h_given_v(self, v):

```

```

# p(h=1|v)
h_prob = torch.sigmoid(F.linear(v, self.W, self.hb))
h_sample = torch.bernoulli(h_prob)
return h_prob, h_sample

def sample_v_given_h(self, h):
    # p(v|h) - Гауссиан с E[v|h] и var=1 (т.к. данные стандартизированы)
    v_mean = F.linear(h, self.W.t(), self.vb)
    # Для CD-k мы обычно не сэмплируем, а используем среднее
    return v_mean, v_mean

def forward(self, v):
    # CD-1 (Contrastive Divergence k=1)
    h0_prob, h0_sample = self.sample_h_given_v(v)
    v1_mean, v1_sample = self.sample_v_given_h(h0_sample)
    h1_prob, _ = self.sample_h_given_v(v1_sample) # h1_sample не нужен для CD-1

    return v, h0_prob, v1_sample, h1_prob

def train_rbm(rbm, data_tensor, epochs=50, lr=1e-3, batch_size=BATCH_SIZE):
    rbm = rbm.to(device)
    ds = TensorDataset(data_tensor)
    loader = DataLoader(ds, batch_size=batch_size, shuffle=True)

    for epoch in range(epochs):
        for (xb,) in loader:
            xb = xb.to(device)

            # CD-1
            v0, h0_prob, v1, h1_prob = rbm(xb)

            # Градиенты
            pos_grad = torch.matmul(h0_prob.t(), v0)
            neg_grad = torch.matmul(h1_prob.t(), v1)

            # Обновление весов (с lr и делением на батч)
            # Мы хотим увеличить P(v0) и уменьшить P(v1)
            rbm.W.data += lr * (pos_grad - neg_grad) / xb.size(0)
            rbm.vb.data += lr * torch.mean(v0 - v1, dim=0)
            rbm.hb.data += lr * torch.mean(h0_prob - h1_prob, dim=0)

    return rbm

print("Pretraining RBMs (Fires)...")
X_train_tensor_f_rbm = X_train_f_t.clone().to(device)
rbms_f = []
current_input_f_rbm = X_train_tensor_f_rbm

for h in HIDDEN_SIZES:
    in_dim = current_input_f_rbm.shape[1]
    rbm = RBM(in_dim, h)

```

```

rbm = train_rbm(rbm, current_input_f_rbm, epochs=EPOCHS_RBM_FIRES,
lr=LR_PRETRAIN_FIRES, batch_size=BATCH_SIZE)
rbms_f.append(rbm)
with torch.no_grad():
    # Передаем *вероятности* активации скрытого слоя на вход следующей RBM
    current_input_f_rbm, _ = rbm.sample_h_given_v(current_input_f_rbm.to(device))
    current_input_f_rbm = current_input_f_rbm.cpu()

# Сборка RBM-MLP
class MLP_Pretrained_RBM(nn.Module):
    def __init__(self, input_dim, rbms, output_dim=1):
        super().__init__()
        layers = []
        prev = input_dim
        for rbm in rbms:
            # RBM W имеет размер (n_hidden, n_visible)
            # Линейный слой PyTorch ожидает (out_features, in_features)
            # Нам нужно W.t() (транспонировать)
            n_out, n_in = rbm.W.shape
            lin_layer = nn.Linear(n_in, n_out)

            # Копируем веса
            lin_layer.weight.data = rbm.W.data.clone() # W уже (out, in)
            lin_layer.bias.data = rbm.hb.data.clone() # Используем скрытый bias

            layers.append(lin_layer)
            layers.append(nn.ReLU())
            prev = n_out

        layers.append(nn.Linear(prev, output_dim)) # Выходной слой
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)

print("Fine-tuning RBM-MLP (Fires)...")
torch.manual_seed(seed)
model_pre_rbm_f = MLP_Pretrained_RBM(input_dim=input_dim_fires, rbms=rbms_f,
output_dim=1).to(device)
model_pre_rbm_f = train_model_regression(
    model_pre_rbm_f, train_loader_f, epochs=EPOCHS_FINE_TUNE_FIRES,
lr=LR_FINE_FIRES, weight_decay=WEIGHT_DECAY
)
metrics_pre_rbm_f_train = evaluate_model_regression(model_pre_rbm_f, X_train_f_t, y_train_f)
metrics_pre_rbm_f_test = evaluate_model_regression(model_pre_rbm_f, X_test_f_t, y_test_f)

# --- Результаты (Fires)
results_fires = {
    "MLP_from_scratch_train": metrics_plain_f_train,
    "MLP_from_scratch_test": metrics_plain_f_test,

```

```

"MLP_pretrained_AE_train": metrics_pre_ae_f_train,
"MLP_pretrained_AE_test": metrics_pre_ae_f_test,
"MLP_pretrained_RBM_train": metrics_pre_rbm_f_train,
"MLP_pretrained_RBM_test": metrics_pre_rbm_f_test,
}
print("\n--- Результаты (Forest Fires) ---")
print(safe_json_dump(results_fires))

# --- Графики (Fires)
labels = ['MLP \n(from scratch)', 'MLP \n(pretrain AE)', 'MLP \n(pretrain RBM)']
x = np.arange(len(labels))
width = 0.25

metrics_train = [metrics_plain_f_train, metrics_pre_ae_f_train, metrics_pre_rbm_f_train]
metrics_test = [metrics_plain_f_test, metrics_pre_ae_f_test, metrics_pre_rbm_f_test]

rmse_train = [m['RMSE'] for m in metrics_train]
rmse_test = [m['RMSE'] for m in metrics_test]
mae_train = [m['MAE'] for m in metrics_train]
mae_test = [m['MAE'] for m in metrics_test]
mape_train = [m['MAPE_%_on_nonzero_targets'] for m in metrics_train]
mape_test = [m['MAPE_%_on_nonzero_targets'] for m in metrics_test]

fig, ax = plt.subplots(1, 3, figsize=(18, 6))
fig.suptitle('Сравнение моделей (Задача Регрессии: Forest Fires)', fontsize=16)

ax[0].bar(x - width/2, rmse_train, width, label='Train')
ax[0].bar(x + width/2, rmse_test, width, label='Test')
ax[0].set_xticks(x)
ax[0].set_xticklabels(labels, rotation=10)
ax[0].set_title('RMSE (меньше - лучше)')
ax[0].legend()

ax[1].bar(x - width/2, mae_train, width, label='Train')
ax[1].bar(x + width/2, mae_test, width, label='Test')
ax[1].set_xticks(x)
ax[1].set_xticklabels(labels, rotation=10)
ax[1].set_title('MAE (меньше - лучше)')
ax[1].legend()

ax[2].bar(x - width/2, mape_train, width, label='Train')
ax[2].bar(x + width/2, mape_test, width, label='Test')
ax[2].set_xticks(x)
ax[2].set_xticklabels(labels, rotation=10)
ax[2].set_title('MAPE on non-zero (%)')
ax[2].legend()

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.savefig("forestfires_metrics_comparison.png", dpi=300)
print("График 'forestfires_metrics_comparison.png' сохранен.")

```

```

plt.show()

#
=====
=====
# --- ЧАСТЬ 2: ЗАДАЧА КЛАССИФИКАЦИИ (RICE)
#
=====
=====
print("\n" + "="*80)
print("--- ЧАСТЬ 2: ЗАДАЧА КЛАССИФИКАЦИИ (RICE) ---")

# --- Предобработка (Rice)
df_rice = load_data(DATA_URL_RICE)

# X - все, кроме 'Class', Y - 'Class'
X_raw_rice = df_rice.drop('Class', axis=1).values.astype(float)
y_str_rice = df_rice['Class'].values

# Кодировем 'Class' (Cammeo, Osmancik) в (0, 1)
le = LabelEncoder()
y_rice = le.fit_transform(y_str_rice)

# Стандартизация X
scaler_rice = StandardScaler()
X_rice = scaler_rice.fit_transform(X_raw_rice)

X_train_r, X_test_r, y_train_r, y_test_r = train_test_split(
    X_rice, y_rice, test_size=0.2, random_state=seed, stratify=y_rice
)

X_train_r_t = torch.tensor(X_train_r, dtype=torch.float32)
X_test_r_t = torch.tensor(X_test_r, dtype=torch.float32)
# Для BCEWithLogitsLoss нужны y типа float и размер [batch, 1]
ytrain_r_t = torch.tensor(y_train_r, dtype=torch.float32).unsqueeze(1)
ytest_r_t = torch.tensor(y_test_r, dtype=torch.float32).unsqueeze(1)

train_ds_r = TensorDataset(X_train_r_t, ytrain_r_t)
test_ds_r = TensorDataset(X_test_r_t, ytest_r_t)
train_loader_r = DataLoader(train_ds_r, batch_size=BATCH_SIZE, shuffle=True)
test_loader_r = DataLoader(test_ds_r, batch_size=BATCH_SIZE, shuffle=False)

input_dim_rice = X_rice.shape[1]

# --- Вспомогательные функции (Классификация)
def train_model_classification(model, train_loader, epochs=100, lr=1e-3, weight_decay=0.0):
    model = model.to(device)
    opt = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=weight_decay)
    # Используем BCEWithLogitsLoss для бинарной классификации (выход 1 logit)
    criterion = nn.BCEWithLogitsLoss()

```

```

for epoch in range(epochs):
    model.train()
    for xb, yb in train_loader:
        xb, yb = xb.to(device), yb.to(device)
        opt.zero_grad()
        out_logits = model(xb)
        loss = criterion(out_logits, yb)
        loss.backward()
        opt.step()
    return model

def evaluate_model_classification(model, X_t, y_true):
    model.eval()
    y_true = y_true.reshape(-1)
    with torch.no_grad():
        logits = model(X_t.to(device)).cpu()
        probs = torch.sigmoid(logits).numpy().reshape(-1)

    preds = (probs > 0.5).astype(int)

    return {
        'Accuracy': accuracy_score(y_true, preds),
        'F1_Score': f1_score(y_true, preds),
        'Precision': precision_score(y_true, preds),
        'Recall': recall_score(y_true, preds)
    }

# --- Модели (Классификация)
# Мы можем переиспользовать классы MLP, MLP_Pretrained_AE, MLP_Pretrained_RBM
# т.к. мы передаем им output_dim=1

# 2.1 MLP (без предобучения)
print("Training MLP from scratch (Rice)...")
torch.manual_seed(seed)
model_plain_r = MLP(input_dim=input_dim_rice, hidden_sizes=HIDDEN_SIZES,
output_dim=1)
model_plain_r = train_model_classification(
    model_plain_r, train_loader_r, epochs=EPOCHS_MLP_RICE, lr=LR_MLP_RICE,
weight_decay=WEIGHT_DECAY
)
metrics_plain_r_train = evaluate_model_classification(model_plain_r, X_train_r_t, y_train_r)
metrics_plain_r_test = evaluate_model_classification(model_plain_r, X_test_r_t, y_test_r)

# 2.2 Autoencoders (предобучение)
print("Pretraining Autoencoders (Rice)...")
X_train_tensor_r = X_train_r_t.clone().to(device)
encoders_r = []
current_input_r = X_train_tensor_r
for h in HIDDEN_SIZES:
    in_dim = current_input_r.shape[1]

```

```

ae = Autoencoder(in_dim, h)
# Используем функции обучения АЕ из Части 1
ae = train_autoencoder(ae, current_input_r, epochs=EPOCHS_AE_RICE,
lr=LR_PRETRAIN_RICE)
with torch.no_grad():
    encoded = ae.encode(current_input_r.to(device)).cpu()
    encoders_r.append(ae.encoder)
    current_input_r = torch.tensor(encoded, dtype=torch.float32)

print("Fine-tuning AE-MLP (Rice)...")
torch.manual_seed(seed)
model_pre_ae_r = MLP_Pretrained_AE(input_dim=input_dim_rice, encoders=encoders_r,
output_dim=1).to(device)
model_pre_ae_r = train_model_classification(
    model_pre_ae_r, train_loader_r, epochs=EPOCHS_FINE_TUNE_RICE, lr=LR_FINE_RICE,
weight_decay=WEIGHT_DECAY
)
metrics_pre_ae_r_train = evaluate_model_classification(model_pre_ae_r, X_train_r_t, y_train_r)
metrics_pre_ae_r_test = evaluate_model_classification(model_pre_ae_r, X_test_r_t, y_test_r)

# 2.3 RBM (предобучение)
print("Pretraining RBMs (Rice)...")
X_train_tensor_r_rbm = X_train_r_t.clone().to(device)
rbms_r = []
current_input_r_rbm = X_train_tensor_r_rbm

for h in HIDDEN_SIZES:
    in_dim = current_input_r_rbm.shape[1]
    rbm = RBM(in_dim, h)
    # Используем функции обучения RBM из Части 1
    rbm = train_rbm(rbm, current_input_r_rbm, epochs=EPOCHS_RBM_RICE,
lr=LR_PRETRAIN_RICE, batch_size=BATCH_SIZE)
    rbms_r.append(rbm)
    with torch.no_grad():
        current_input_r_rbm, _ = rbm.sample_h_given_v(current_input_r_rbm.to(device))
        current_input_r_rbm = current_input_r_rbm.cpu()

print("Fine-tuning RBM-MLP (Rice)...")
torch.manual_seed(seed)
model_pre_rbm_r = MLP_Pretrained_RBM(input_dim=input_dim_rice, rbms=rbms_r,
output_dim=1).to(device)
model_pre_rbm_r = train_model_classification(
    model_pre_rbm_r, train_loader_r, epochs=EPOCHS_FINE_TUNE_RICE, lr=LR_FINE_RICE,
weight_decay=WEIGHT_DECAY
)
metrics_pre_rbm_r_train = evaluate_model_classification(model_pre_rbm_r, X_train_r_t,
y_train_r)
metrics_pre_rbm_r_test = evaluate_model_classification(model_pre_rbm_r, X_test_r_t, y_test_r)

# --- Результаты (Rice)

```

```

results_rice = {
    "MLP_from_scratch_train": metrics_plain_r_train,
    "MLP_from_scratch_test": metrics_plain_r_test,
    "MLP_pretrained_AE_train": metrics_pre_ae_r_train,
    "MLP_pretrained_AE_test": metrics_pre_ae_r_test,
    "MLP_pretrained_RBM_train": metrics_pre_rbm_r_train,
    "MLP_pretrained_RBM_test": metrics_pre_rbm_r_test,
}
print("\n--- Результаты (Rice Classification) ---")
print(safe_json_dump(results_rice))

# --- Графики (Rice)
labels_r = ['MLP \n(from scratch)', 'MLP \n(pretrain AE)', 'MLP \n(pretrain RBM)']
x_r = np.arange(len(labels_r))
width_r = 0.25

metrics_train_r = [metrics_plain_r_train, metrics_pre_ae_r_train, metrics_pre_rbm_r_train]
metrics_test_r = [metrics_plain_r_test, metrics_pre_ae_r_test, metrics_pre_rbm_r_test]

acc_train = [m['Accuracy'] for m in metrics_train_r]
acc_test = [m['Accuracy'] for m in metrics_test_r]
f1_train = [m['F1_Score'] for m in metrics_train_r]
f1_test = [m['F1_Score'] for m in metrics_test_r]

fig_r, ax_r = plt.subplots(1, 2, figsize=(14, 6))
fig_r.suptitle('Сравнение моделей (Задача Классификации: Rice)', fontsize=16)

ax_r[0].bar(x_r - width_r/2, acc_train, width_r, label='Train')
ax_r[0].bar(x_r + width_r/2, acc_test, width_r, label='Test')
ax_r[0].set_xticks(x_r)
ax_r[0].set_xticklabels(labels_r, rotation=10)
ax_r[0].set_title('Accuracy (больше - лучше)')
ax_r[0].set_ylabel('Accuracy')
ax_r[0].set_ylim(0.9, 1.0) # Масштабируем для наглядности
ax_r[0].legend()

ax_r[1].bar(x_r - width_r/2, f1_train, width_r, label='Train')
ax_r[1].bar(x_r + width_r/2, f1_test, width_r, label='Test')
ax_r[1].set_xticks(x_r)
ax_r[1].set_xticklabels(labels_r, rotation=10)
ax_r[1].set_title('F1 Score (больше - лучше)')
ax_r[1].set_ylabel('F1 Score')
ax_r[1].set_ylim(0.9, 1.0) # Масштабируем для наглядности
ax_r[1].legend()
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.savefig("rice_metrics_comparison.png", dpi=300)
print("График 'rice_metrics_comparison.png' сохранен.")
plt.show() # Показать оба графика в конце

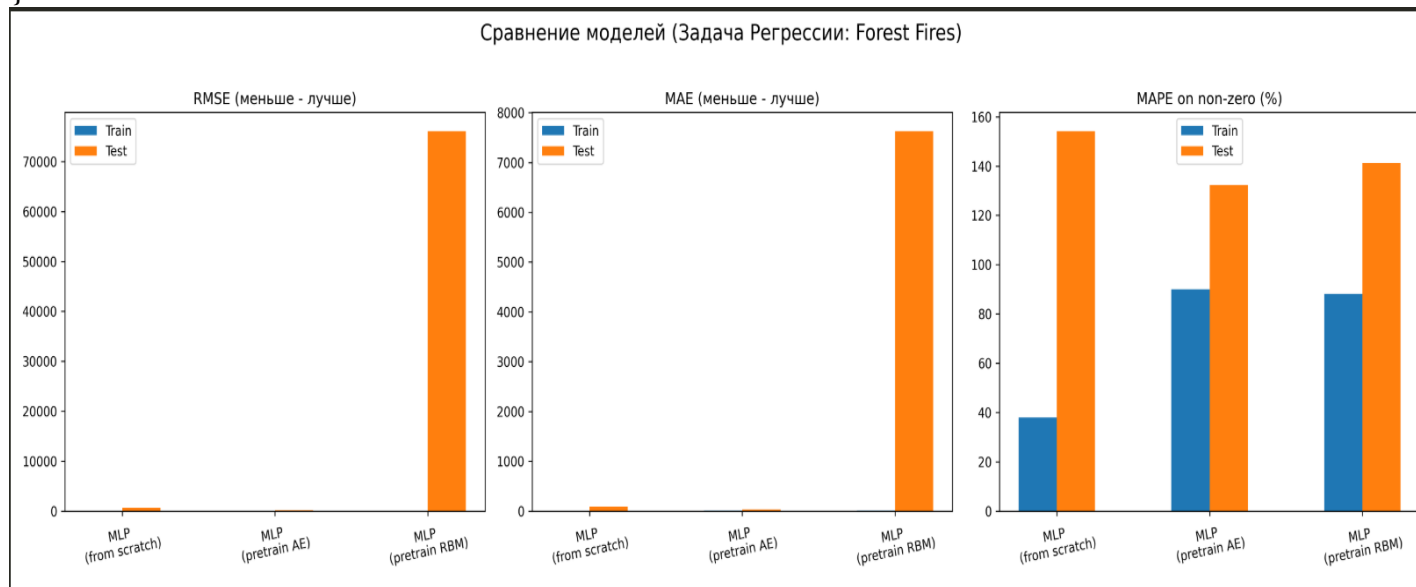
print("\n--- Выполнение завершено. ---")

```

## Часть 1. Задача регрессии(forest fires)

--- Результаты (Forest Fires) ---

```
{
  "MLP_from_scratch_train": {
    "RMSE": 13.085612774385318,
    "MAE": 3.3376202619413573,
    "MAPE_%_on_nonzero_targets": 37.97687956599512
  },
  "MLP_from_scratch_test": {
    "RMSE": 647.4612876513439,
    "MAE": 86.15438340669124,
    "MAPE_%_on_nonzero_targets": 154.1733287299697
  },
  "MLP_pretrained_AE_train": {
    "RMSE": 42.0328119662878,
    "MAE": 8.817069362911596,
    "MAPE_%_on_nonzero_targets": 89.90347870098614
  },
  "MLP_pretrained_AE_test": {
    "RMSE": 135.2418721392766,
    "MAE": 28.059882161161646,
    "MAPE_%_on_nonzero_targets": 132.27002726060098
  },
  "MLP_pretrained_RBM_train": {
    "RMSE": 41.04658729174721,
    "MAE": 8.226596767560332,
    "MAPE_%_on_nonzero_targets": 88.11238343650174
  },
  "MLP_pretrained_RBM_test": {
    "RMSE": 76115.62564691626,
    "MAE": 7629.771024180647,
    "MAPE_%_on_nonzero_targets": 141.21044283543196
  }
}
```

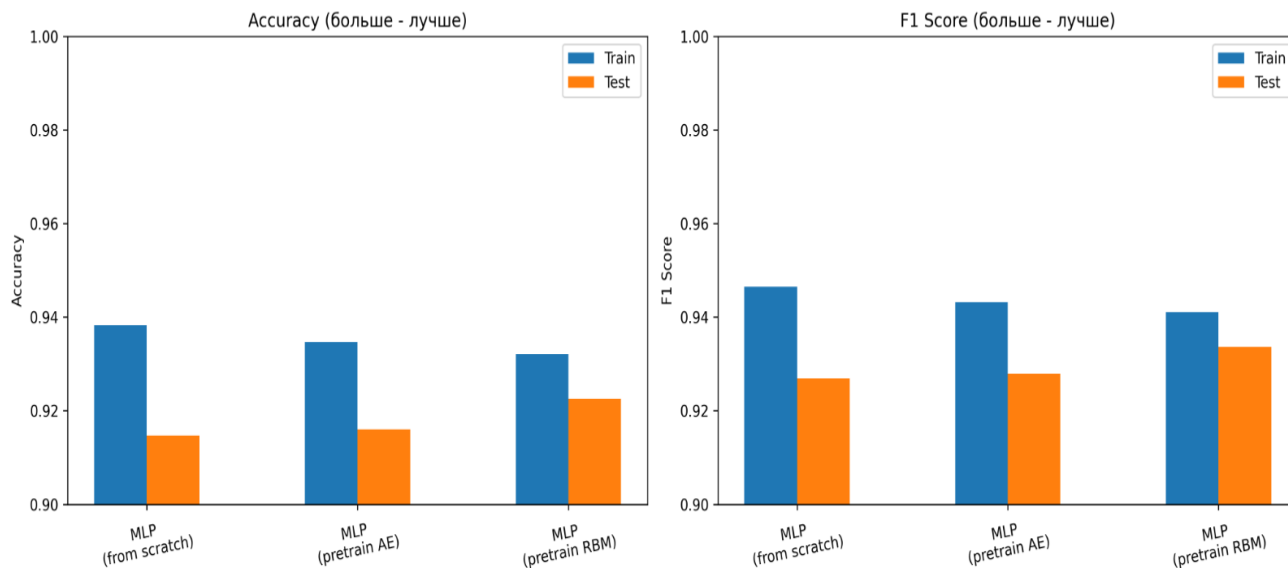


## Часть 2. Задача классификации(rise)

--- Результаты (Rice Classification) ---

```
{
  "MLP_from_scratch_train": {
    "Accuracy": 0.9383202099737533,
    "F1_Score": 0.9464997154240182,
    "Precision": 0.93954802259887,
    "Recall": 0.9535550458715596
  },
  "MLP_from_scratch_test": {
    "Accuracy": 0.9146981627296588,
    "F1_Score": 0.9268841394825647,
    "Precision": 0.9094922737306843,
    "Recall": 0.944954128440367
  },
  "MLP_pretrained_AE_train": {
    "Accuracy": 0.9347112860892388,
    "F1_Score": 0.9431915500999144,
    "Precision": 0.939169982944855,
    "Recall": 0.9472477064220184
  },
  "MLP_pretrained_AE_test": {
    "Accuracy": 0.916010498687664,
    "F1_Score": 0.9279279279279279,
    "Precision": 0.911504424778761,
    "Recall": 0.944954128440367
  },
  "MLP_pretrained_RBM_train": {
    "Accuracy": 0.9320866141732284,
    "F1_Score": 0.9411095305832148,
    "Precision": 0.9339356295878035,
    "Recall": 0.948394495412844
  },
  "MLP_pretrained_RBM_test": {
    "MLP_pretrained_RBM_test": {
      "Accuracy": 0.9225721784776902,
      "MLP_pretrained_RBM_test": {
        "MLP_pretrained_RBM_test": {
          "MLP_pretrained_RBM_test": {
            "MLP_pretrained_RBM_test": {
              "Accuracy": 0.9225721784776902,
              "MLP_pretrained_RBM_test": {
                "Accuracy": 0.9225721784776902,
                "MLP_pretrained_RBM_test": {
                  "Accuracy": 0.9225721784776902,
                  "F1_Score": 0.9336332958380202,
                  "Precision": 0.9161147902869757,
                  "Recall": 0.9518348623853211
                }
              }
            }
          }
        }
      }
    }
  }
}
```

## Сравнение моделей (Задача Классификации: Rice)



Метод	Плюсы	Минусы
MLP (с нуля)	Простота реализации, быстрое обучение.	Может застрять в плохом локальном минимуме на сложных, глубоких сетях.
MLP + Autoencoder	Хорошая инициализация весов. Концептуально прост (обучение на восстановление).	Требует доп. времени на послышное обучение.
MLP + RBM	Теоретически более мощный (генеративная модель), чем AE (дискриминативная).	Сложнее в реализации (CD-k), чувствителен к lr. На практике (особенно с ReLU-сетями) часто не дает преимуществ перед AE или хорошей инициализацией (He/Xavier).

**Вывод:** Изучил RBM и разработал нейросеть для задачи регрессии и классификации.