CSCI/MATH-485

Assignment-4

Snehitha Gorantla

04-08-2025

PERCEPTRON

Introduction

This report presents the implementation and analysis of two perceptron algorithms: the heuristic approach of perceptron and a perceptron with gradient descent approach. The experiments evaluate the performance of both algorithms with varying learning rates (0.01, 0.1, and 1.0) on a binary classification task.

Data Loading and Visualization

The dataset was loaded from a CSV file and split into features (X) and labels (y):

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
```

✓ 0.0s

```
data = pd.read_csv("C:/Users/snehi/Downloads/data.csv", header=None)
X = data.iloc[:, 0:2].values
y = data.iloc[:, 2].values
```

✓ 0.0s

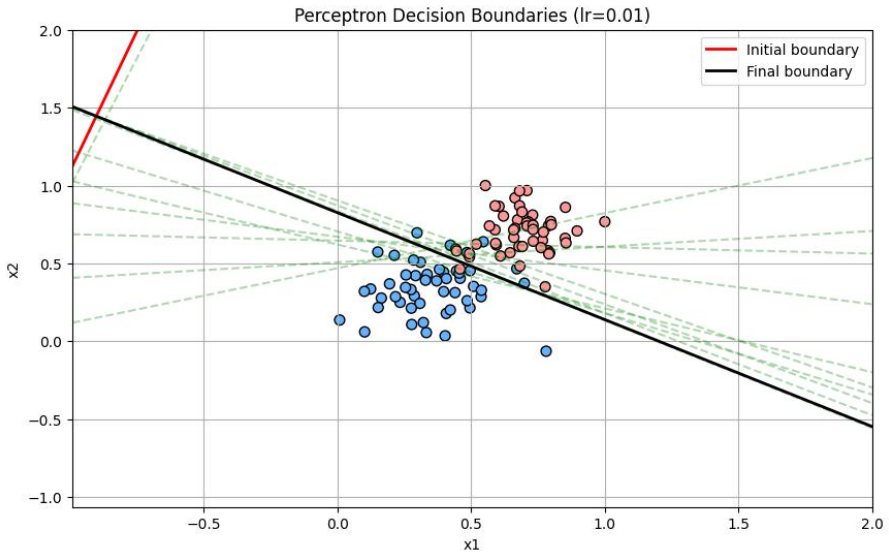
Part-1: Implementing Perceptron Using Heuristic Approach

```python
def perceptron(X, y, learning_rate=0.1, max_iterations=65, plot_steps=False):
    n_features = X.shape[1]
    np.random.seed(42)
    weights = np.random.randn(n_features)
    bias = np.random.randn()

    history = [(np.copy(weights), bias)]

    iteration = 0
    for _ in range(max_iterations):
        iteration += 1
        misclassified = 0

        for i in range(len(X)):
            z = np.dot(X[i], weights) + bias
            prediction = 1 if z >= 0 else 0

            if prediction != y[i]:
                misclassified += 1

                if prediction == 0:  # Actual is 1, prediction is 0
                    bias += learning_rate
                    weights += learning_rate * X[i]
                else:  # Actual is 0, prediction is 1
                    bias -= learning_rate
                    weights -= learning_rate * X[i]

                history.append((np.copy(weights), bias))

        if misclassified == 0:
            print(f"Converged after {iteration} iterations")
            break

    return weights, bias, history, iteration
```

```python
def experiment_with_boundary_evolution(X, y, learning_rates=[0.01, 0.1, 1], max_iterations=65):
    results = {}
    for lr in learning_rates:
        print(f"\nTraining with learning rate: {lr}")
        weights, bias, history, iterations = perceptron(X, y, learning_rate=lr,
                                                        max_iterations=max_iterations)
        correct = 0
        for i in range(len(X)):
            z = np.dot(X[i], weights) + bias
            prediction = 1 if z >= 0 else 0
            if prediction == y[i]:
                correct += 1
        accuracy = correct / len(X) * 100
        results[lr] = {
            'weights': weights,
            'bias': bias,
            'iterations': iterations,
            'accuracy': accuracy,
            'history': history
        }
        print(f"Learning rate: {lr}")
        print(f"Iterations to converge: {iterations}")
        print(f"Final weights: {weights}")
        print(f"Final bias: {bias}")
        print(f"Accuracy: {accuracy:.2f}%")

        plot_decision_boundaries(X, y, history, lr)
    return results
results = experiment_with_boundary_evolution(X, y, learning_rates=[0.01, 0.1, 1.0])
plt.show()
```

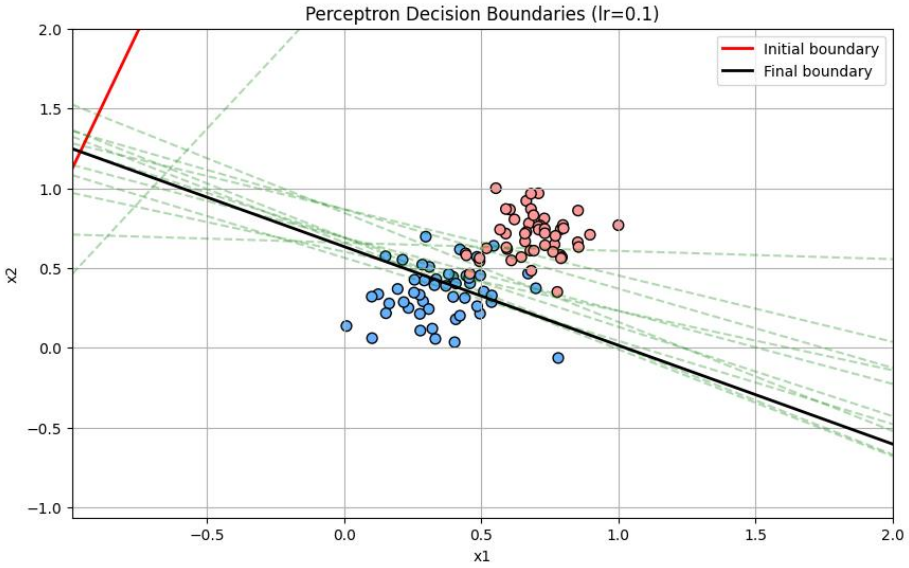
Training with learning rate: 0.01

- Learning rate: 0.01
- Iterations to converge: 65
- Final weights: [-0.18100855 -0.263552]
- Final bias: 0.2176885381006921
- Accuracy: 93.00%



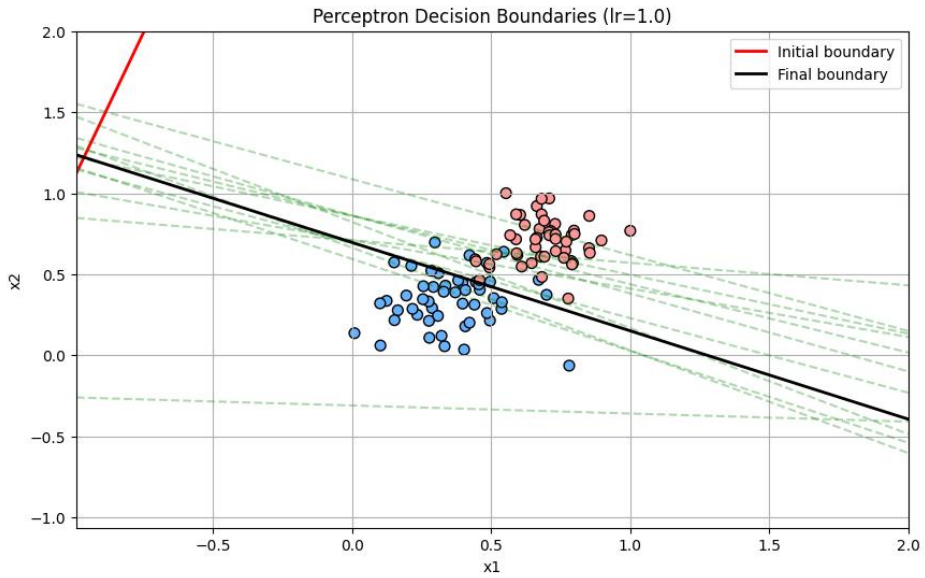
Training with learning rate: 0.1

- Learning rate: 0.1
- Iterations to converge: 65
- Final weights: [-0.53425085 -0.8637153]
- Final bias: 0.5476885381006925
- Accuracy: 80.00%



Training with learning rate: 1.0

- Learning rate: 1.0
- Iterations to converge: 65
- Final weights: [-4.42335585 -8.1133413]
- Final bias: 5.6476885381006925
- Accuracy: 93.00%



Results Analysis

The standard perceptron algorithm showed interesting behavior across different learning rates:

1. **Learning Rate 0.01 (Low)**:
   - Achieved high accuracy (93.00%)
   - The weights and bias are relatively small, suggesting a gentle decision boundary.
   - The algorithm takes smaller steps during training, resulting in a more gradual evolution of the decision boundary.
   - The small weight magnitudes indicate that the model is less sensitive to minor changes in input features.
2. **Learning Rate 0.1 (Medium)**:
   - Achieved the lowest accuracy (80.00%) among the three rates.
   - This suggests that this particular learning rate might have caused the algorithm to miss the optimal solution by taking steps that were either too large to fine-tune or too small to escape local minima.
   - The weights are larger in magnitude than with LR=0.01 but smaller than with LR=1.0
   - The decision boundary evolution shows more significant shifts between iterations.

3. **Learning Rate 1.0 (High)**:
    - Also achieved high accuracy (93.00%), matching the performance of LR=0.01
    - The weights and bias are significantly larger in magnitude, indicating a steeper decision boundary.
    - This suggests that the model makes more dramatic updates with each misclassification.
    - While effective for this dataset, such large steps might cause instability in more complex problems.

Part 2: Perceptron with Gradient Descent

Implementation

The perceptron with gradient descent uses the sigmoid function as an activation function and optimizes weights using gradient descent to minimize log loss:

```python
# Sigmoid activation function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Log loss (binary cross-entropy) function
def log_loss(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
```

```python
# Function to plot the error (log loss) over epochs
def plot_error_graph(errors, learning_rate, epoch_interval=5):
    plt.figure(figsize=(10, 6))
    epochs = np.arange(0, len(errors) * epoch_interval, epoch_interval)

    plt.plot(epochs, errors, marker='o')
    plt.xlabel('Epochs')
    plt.ylabel('Log Loss')
    plt.title(f'Log Loss vs. Epochs (lr={learning_rate})')
    plt.grid(True)
    plt.show()
```

```python
def perceptron_gradient_descent(X, y, learning_rate=0.1, epochs=100, plot_interval=10):
    n_features = X.shape[1]
    np.random.seed(42)
    weights = np.random.randn(n_features)
    bias = np.random.randn()
    history = [(np.copy(weights), bias)]
    errors = []

    for epoch in range(epochs):
        z = np.dot(X, weights) + bias
        y_pred = sigmoid(z)
        if epoch % plot_interval == 0:
            current_loss = log_loss(y, y_pred)
            errors.append(current_loss)
            print(f"Epoch {epoch}, Log Loss: {current_loss:.6f}")
        # Backward pass - update weights and bias
        error = y - y_pred
        # Update bias: b + r(y - ŷ) → b
        bias += learning_rate * np.sum(error)
        # Update weights: wi + r(y - ŷ)xi → wi
        for i in range(n_features):
            weights[i] += learning_rate * np.sum(error * X[:, i])

        if epoch % plot_interval == 0:
            history.append((np.copy(weights), bias))

    z = np.dot(X, weights) + bias
    y_pred = sigmoid(z)
    final_loss = log_loss(y, y_pred)
    print(f"Final Log Loss: {final_loss:.6f}")
    last_weights, last_bias = history[-1]
    if not np.array_equal(weights, last_weights) or bias != last_bias:
        history.append((np.copy(weights), bias))
    return weights, bias, history, errors
```

```python
def experiment_learning_rates(X, y, learning_rates=[0.01, 0.1, 1], epochs=100, plot_interval=10):
    results = {}
    for lr in learning_rates:
        print(f"\nTraining with learning rate: {lr}")
        weights, bias, history, errors = perceptron_gradient_descent(
            X, y, learning_rate=lr, epochs=epochs, plot_interval=plot_interval
        )
        z = np.dot(X, weights) + bias
        y_pred = sigmoid(z)
        y_pred_binary = (y_pred >= 0.5).astype(int)
        accuracy = np.mean(y_pred_binary == y) * 100
        results[lr] = {
            'weights': weights,
            'bias': bias,
            'accuracy': accuracy,
            'errors': errors,
            'history': history
        }
        print(f"Learning rate: {lr}")
        print(f"Final weights: {weights}")
        print(f"Final bias: {bias}")
        print(f"Accuracy: {accuracy:.2f}%")
        plot_decision_boundaries(X, y, history, lr, plot_interval)
        plot_error_graph(errors, lr, plot_interval)
    return results
print("Perceptron with Gradient Descent")
plot_interval = 10
results = experiment_learning_rates(
    X, y,
    learning_rates=[0.01, 0.1, 1.0],
    epochs=100,
    plot_interval=plot_interval
)
plt.show()
```

Training with learning rate: 0.01

Epoch 0, Log Loss: 0.807333

Epoch 10, Log Loss: 0.588202

Epoch 20, Log Loss: 0.527992

Epoch 30, Log Loss: 0.480655

Epoch 40, Log Loss: 0.442831

Epoch 50, Log Loss: 0.412104

Epoch 60, Log Loss: 0.386743

Epoch 70, Log Loss: 0.365503

Epoch 80, Log Loss: 0.347481

Epoch 90, Log Loss: 0.332010

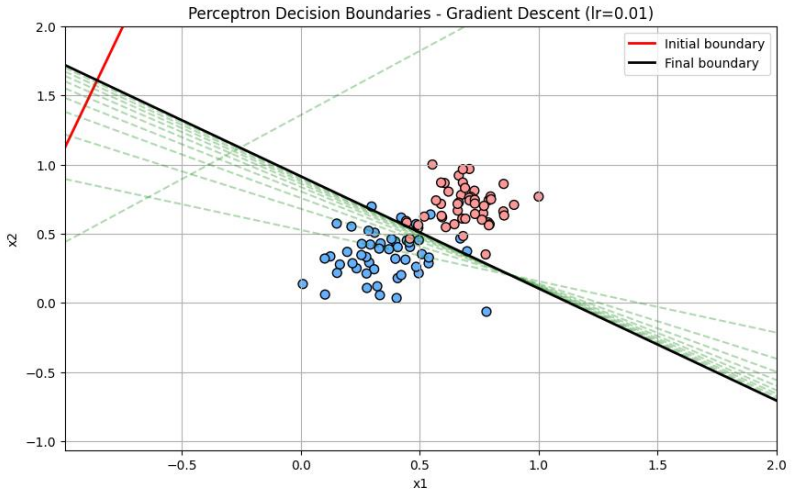
Final Log Loss: 0.318591

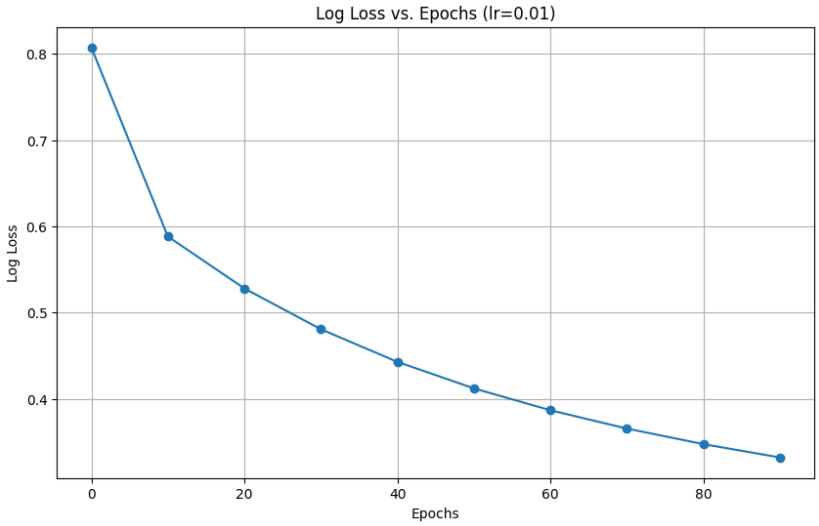
Learning rate: 0.01

Final weights: [-3.02581855 -3.73366866]

Final bias: 3.4151735371848457

Accuracy: 93.00%





Training with learning rate: 0.1

Epoch 0, Log Loss: 0.807333

Epoch 10, Log Loss: 0.512779

Epoch 20, Log Loss: 0.203829

Epoch 30, Log Loss: 0.186354

Epoch 40, Log Loss: 0.176658

Epoch 50, Log Loss: 0.169742

Epoch 60, Log Loss: 0.164568

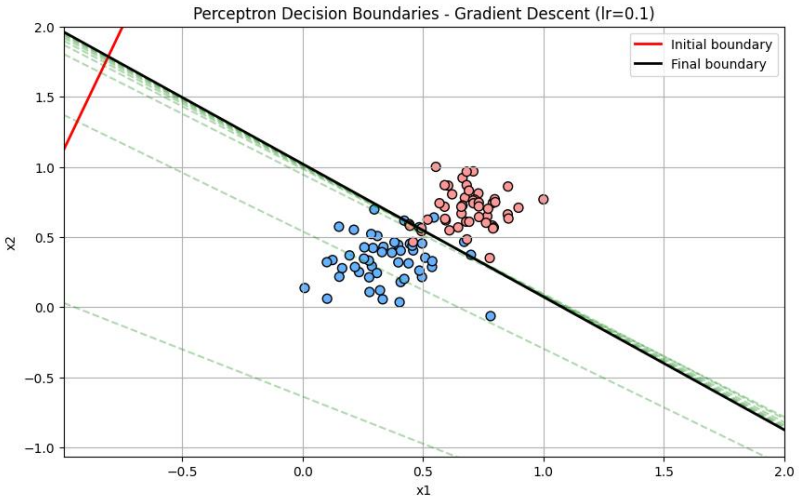
Epoch 70, Log Loss: 0.160559

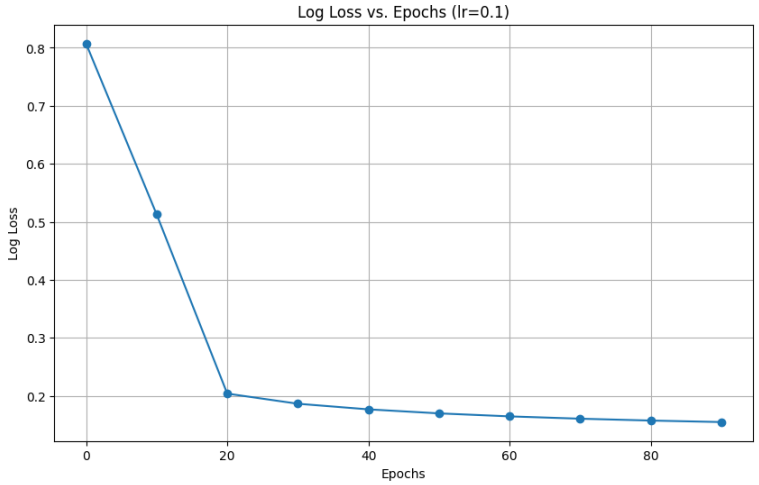
Epoch 80, Log Loss: 0.157368

Epoch 90, Log Loss: 0.154773

Final Log Loss: 0.152626

- Learning rate: 0.1
- Final weights: [-9.40680912 -9.92431046]
- Final bias: 10.136624599036498
- Accuracy: 92.00%





Training with learning rate: 1.0

Epoch 0, Log Loss: 0.807333

Epoch 10, Log Loss: 11.453220

Epoch 20, Log Loss: 1.047947

Epoch 30, Log Loss: 0.330794

Epoch 40, Log Loss: 0.345697

Epoch 50, Log Loss: 0.349333

Epoch 60, Log Loss: 0.349337

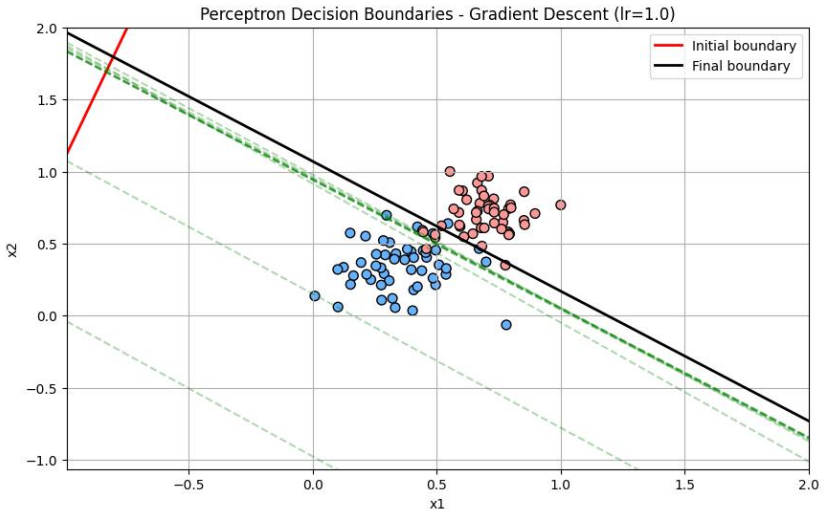
Epoch 70, Log Loss: 0.348165

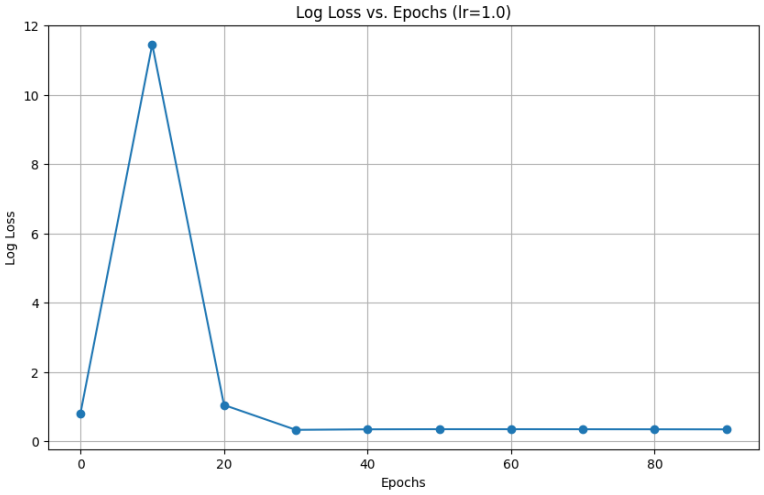
Epoch 80, Log Loss: 0.346541

Epoch 90, Log Loss: 0.344759

Final Log Loss: 0.342971

- Learning rate: 1.0
- Final weights: [-49.4253871 -54.88688276]
- Final bias: 58.76764766022309
- Accuracy: 93.00%





Results Analysis

The perceptron with gradient descent showed distinct behavior across different learning rates:

1. **Learning Rate 0.01 (Low)**:
    - Achieved high accuracy (93.00%)
    - The loss decreased steadily but slowly, as seen in the loss curve.
    - Starting with Log Loss of 0.807333 and ending at 0.318591
    - The final weights and bias are moderate in magnitude.

- The loss decreased consistently across epochs, with significant improvements early on (from epoch 0 to 30) and more gradual improvements later.

2. **Learning Rate 0.1 (Medium)**:
   - Achieved 92.00% accuracy, slightly lower than the other two rates.
   - However, it achieved the lowest log loss (0.152626), indicating better probability calibration.
   - The loss decreased more rapidly than with LR=0.01
   - The log loss decreased dramatically in the first 20 epochs (from 0.807333 to 0.203829)
   - The weights and bias are larger in magnitude compared to LR=0.01
   - The faster convergence suggests this is a more efficient learning rate for this problem.
3. **Learning Rate 1.0 (High)**:
   - Achieved high accuracy (93.00%)
   - The loss curve shows initial instability (a spike at around epoch 10 with log loss of 11.453220)
   - Eventually stabilized with a final log loss of 0.342971
   - After the initial instability, the loss decreased rapidly between epochs 10 and 30.
   - The weights and bias are significantly larger in magnitude, suggesting a very steep decision boundary.
   - The instability indicates that the learning rate is possibly too high, causing the model to overshoot the optimal solution initially.

The loss curves:

- LR=0.01: Smooth, steady decrease in loss with no instability
- LR=0.1: Faster decrease with good final convergence and lowest final loss
- LR=1.0: Initial instability followed by recovery and convergence, but with higher final loss than LR=0.1