



# Machine Learning

## LABORATORY: Regularization Homework

NAME:

STUDENT ID#:

### Objectives:

- Understand the concept of regularization and its importance in preventing overfitting.
- Implement and compare two regularization strategies:
  - Early Stopping (validation-based regularization)
  - Weight Decay (L2 regularization)
- Apply these methods to a binary classification task using the MNIST dataset.
- Visualize and interpret training/validation loss and accuracy curves.
- Analyze model behavior by examining misclassified test samples.

### Part 1. Instruction

- In this assignment, use the existing neural network structure.
  - A single hidden layer with ReLU activation
  - Softmax output for multi-class classification (later converted to binary)
- Implement and compare:
  - **Early Stopping:** Stop training if the validation loss does not improve for several epochs
  - **Weight Decay (L2 Regularization):** Penalize large weights by adding a regularization term to the loss function
- You may write all algorithms in **one file with selectable modes**, or in **three separate files**.
- Do not use external machine learning libraries (e.g., scikit-learn, PyTorch).
- For each method (early stopping and weight decay):
  - Plot the training vs validation loss curves
  - Plot the training vs validation accuracy curves



## Part 2. Code Template

Step	Procedure
1	<pre> # ===== Load Dataset ===== def load_images(filename):     with open(filename, 'rb') as f:         _, num, rows, cols = struct.unpack("&gt;IIII", f.read(16))         data = np.frombuffer(f.read(), dtype=np.uint8).reshape((num, rows * cols))         return data.astype(np.float32) / 255.0  def load_labels(filename):     with open(filename, 'rb') as f:         _, num = struct.unpack("&gt;II", f.read(8))         return np.frombuffer(f.read(), dtype=np.uint8)         return labels[:num] </pre>
2	<pre> # TODO: Complete all the functions, you may change the structures # ===== 2. Utils ===== def shuffle_numpy(X, y):     pass  def split_train_val(X, y, val_ratio=0.2):     pass  def one_hot(y, num_classes):     pass  def accuracy(Y_pred, Y_true):     pass  # ===== 3. Model ===== class MLP:     def __init__(self, input_dim, hidden_dim, output_dim, weight_decay=0.0):         self.W1 = np.random.randn(input_dim, hidden_dim) * 0.01         self.b1 = np.zeros((1, hidden_dim))         self.W2 = np.random.randn(hidden_dim, output_dim) * 0.01         self.b2 = np.zeros((1, output_dim))         self.lambda_ = weight_decay      def relu(self, x):         return np.maximum(0, x)      def relu_deriv(self, x):         return (x &gt; 0).astype(float)      def softmax(self, x):         exps = np.exp(x - np.max(x, axis=1, keepdims=True))         return exps / np.sum(exps, axis=1, keepdims=True) </pre>



```

def forward(self, X):
    self.z1 = X @ self.W1 + self.b1
    self.a1 = self.relu(self.z1)
    self.z2 = self.a1 @ self.W2 + self.b2
    self.a2 = self.softmax(self.z2)
    return self.a2

def compute_loss(self, Y_pred, Y_true):
    TODO: Weight Decay (L2 Regularization)
    pass

def backward(self, X, Y_true, Y_pred, lr=0.1):
    m = Y_true.shape[0]
    dz2 = (Y_pred - Y_true) / m
    dW2 = self.a1.T @ dz2 + self.lambda_ * self.W2
    db2 = np.sum(dz2, axis=0, keepdims=True)
    da1 = dz2 @ self.W2.T
    dz1 = da1 * self.relu_deriv(self.z1)
    dW1 = X.T @ dz1 + self.lambda_ * self.W1
    db1 = np.sum(dz1, axis=0, keepdims=True)

    self.W2 -= lr * dW2
    self.b2 -= lr * db2
    self.W1 -= lr * dW1
    self.b1 -= lr * db1

# ===== 4. Train Function =====
def train(model, X_train, y_train, X_val, y_val, lr=0.1,
epochs=100, use_early_stopping=False, patience=5):
    train_losses, val_losses, train_accs, val_accs = [], [],
    [], []
    best_val_loss = np.inf
    patience_count = 0

    for epoch in range(epochs):
        TODO: complete this part

        print(f"Epoch {epoch:02d} | Train Loss: {loss:.4f} |
Val Loss: {val_loss:.4f}")

        TODO: implement your early stopping strategy here

    return train_losses, val_losses, train_accs, val_accs

```

```

3 # ===== 5. Plotting =====
def plot_curves(train_losses, val_losses, train_accs, val_accs,

```



	<pre> title):     plt.figure(figsize=(12, 5))     plt.subplot(1, 2, 1)     plt.plot(train_losses, label="Train Loss")     plt.plot(val_losses, label="Val Loss")     plt.title("Loss Curve - " + title)     plt.xlabel("Epochs")     plt.ylabel("Loss")     plt.legend()      plt.subplot(1, 2, 2)     plt.plot(train_accs, label="Train Acc")     plt.plot(val_accs, label="Val Acc")     plt.title("Accuracy Curve - " + title)     plt.xlabel("Epochs")     plt.ylabel("Accuracy")     plt.legend()     plt.tight_layout()     plt.show() </pre>
4	<pre> # ===== 6. Main ===== if __name__ == "__main__":     X = load_images("train-images.idx3-ubyte")     y = load_labels("train-labels.idx1-ubyte")     X, y = shuffle_numpy(X, y)     X_train, y_train, X_val, y_val = split_train_val(X, y)     y_train_oh = one_hot(y_train, 10)     y_val_oh = one_hot(y_val, 10)      # === OPTION 1: Early Stopping ===     # model_early = MLP(_, _, _, weight_decay=0.0)     # t1, v1, a1, a2 = train(model_early, X_train, y_train_oh,     X_val, y_val_oh, use_early_stopping=True)     # plot_curves(t1, v1, a1, a2, title="Early Stopping")      # === OPTION 2: Weight Decay === </pre>

## Grading Assignment & Submission (70% Max)

### Implementation (50%):

Correctly implemented, runs, and shows the plotting result for:

- **(20%) Early Stopping**
  - Uses validation loss to stop training early
- **(20%) Weight Decay (L2 Regularization)**
  - Applies L2 penalty to loss and gradients
- **(10%) Comparison**



Visualizes and compares the performance of both techniques (**Please provide simple discussion of your result**)

Includes:

- Training vs validation curves
- Result of 3 different  $\lambda$  value

**Question (20%):**

**1. (7%) Which regularization method gave you the best test accuracy?**

Why do you think it performed better than the other? Was it due to training duration, generalization effect, or another factor?

**2. (7%) Compare training and validation loss curves**

Which method showed signs of overfitting or underfitting?

Use your graphs to justify your answer (e.g., early stopping curve flattens early, weight decay trains longer but smoother).

**3. (6%) How did your choice of regularization strength ( $\lambda$ ) or patience affect the model?**

What  $\lambda$  or patience value worked best in your experiment? What happened when you increased or decreased it?

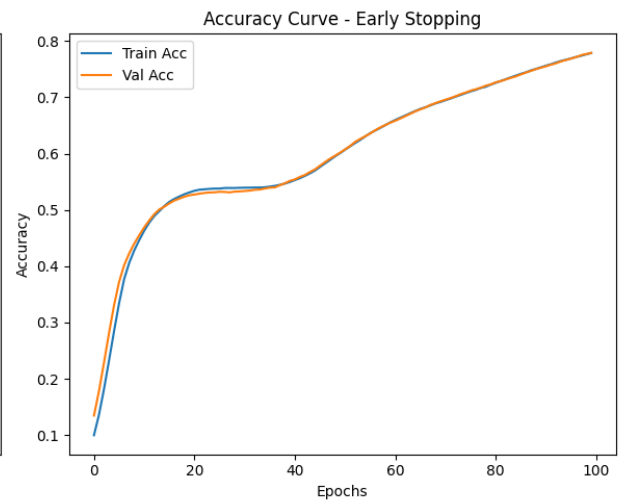
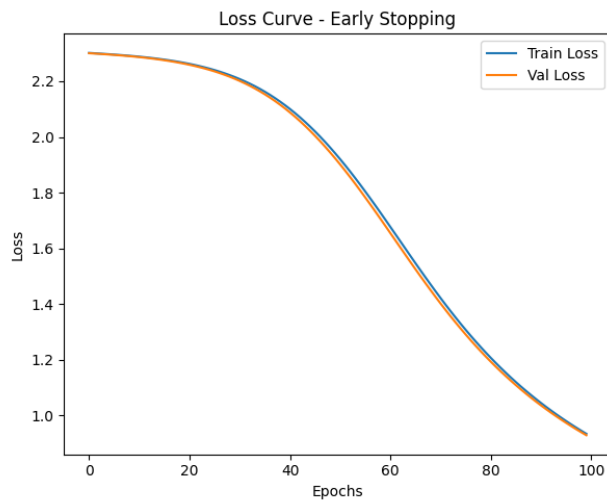
**Submission:**

1. Report: Answer all the questions. Include screenshots of your results and discussion in the last pages of this PDF File.
2. Code: Submit your complete Python script in either .py or .ipynb format.
3. Upload both your report and code to the E3 system (**Labs5 Homework Assignment**). Name your files correctly:
  - a. Report: StudentID\_Lab5\_Homework.pdf
  - b. Code: StudentID\_Lab5\_Homework.py or StudentID\_Lab5\_Homework.ipynb
4. Deadline: Sunday, 21:00 PM
5. Plagiarism is **strictly prohibited**. Submitting copied work from other students will result in penalties.

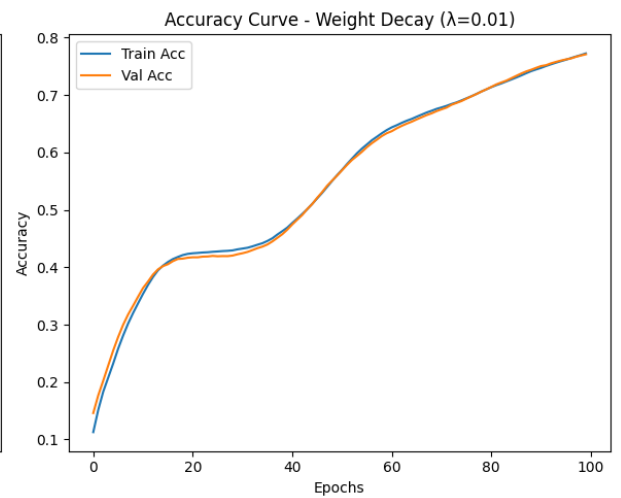
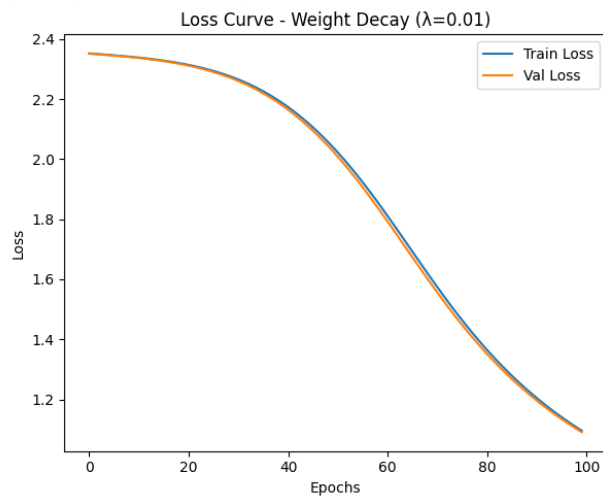
**Example Output (Just for reference):**



Epoch 91	Train Loss: 1.0306	Val Loss: 1.0224
Epoch 92	Train Loss: 1.0173	Val Loss: 1.0095
Epoch 93	Train Loss: 1.0044	Val Loss: 0.9970
Epoch 94	Train Loss: 0.9918	Val Loss: 0.9848
Epoch 95	Train Loss: 0.9796	Val Loss: 0.9730
Epoch 96	Train Loss: 0.9677	Val Loss: 0.9615
Epoch 97	Train Loss: 0.9562	Val Loss: 0.9503
Epoch 98	Train Loss: 0.9450	Val Loss: 0.9395
Epoch 99	Train Loss: 0.9341	Val Loss: 0.9289



Epoch 90	Train Loss: 1.2038	Val Loss: 1.1954
Epoch 91	Train Loss: 1.1903	Val Loss: 1.1823
Epoch 92	Train Loss: 1.1772	Val Loss: 1.1696
Epoch 93	Train Loss: 1.1644	Val Loss: 1.1573
Epoch 94	Train Loss: 1.1521	Val Loss: 1.1454
Epoch 95	Train Loss: 1.1401	Val Loss: 1.1338
Epoch 96	Train Loss: 1.1285	Val Loss: 1.1225
Epoch 97	Train Loss: 1.1172	Val Loss: 1.1116
Epoch 98	Train Loss: 1.1063	Val Loss: 1.1010
Epoch 99	Train Loss: 1.0956	Val Loss: 1.0907



## Results and Discussion:

