Q3 - Neural Networks

در این تمرین شما موارد زیر را انجام خواهید داد:

1. مشتق‌گیری forward and backward propagation
2. پیاده‌سازی یک شبکه عصبی از ابتدا
3. اجرای آزمایش‌هایی با مدل خود

دستور العمل‌ها

در نهایت دفترچه را به صورت فایل پی دی اف ذخیره کنید
(منو File → Print → Save as PDF)

Q1: سوال های حل کردنی

Consider a simple neural network with three layers: an input layer, a hidden layer, and an output layer.

Let $w^{(1)}$ and $w^{(2)}$ be the layers' weight matrices and let $b^{(1)}$ and $b^{(2)}$ be their biases. For convention, suppose that w_{ij} is the weight between the ith node in the previous layer and the jth node in the current one.

Additionally, the activation function for both layers is the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. Let $z^{(1)}$ and $z^{(2)}$ be the outputs of the two layers before activation, and let $a^{(1)} = \sigma(z^{(1)})$ and $a^{(2)} = \sigma(z^{(2)})$.

Lastly, we choose the L2 loss $L(y_{\text{true}}, y_{\text{predict}}) = \frac{1}{2}(y_{\text{true}} - y_{\text{predict}})^2$ as the loss function.

Q1.1: Forward Pass

Suppose that

$$w^{(1)} = \begin{bmatrix} 0.4 & 0.6 & 0.2 \\ 0.3 & 0.9 & 0.5 \end{bmatrix}, b^{(1)} = [1, 1, 1]; \text{ and}$$

$$w^{(2)} = \begin{bmatrix} 0.2 \\ 0.2 \\ 0.8 \end{bmatrix}, b^{(2)} = [0.5].$$

If the input is $a^{(0)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, what is the network output? Show your calculation steps and round your **final** answer to 2 digits after decimal.

[Answer]

Neural Network Architecture & Calculation Explanation

1. Network Architecture

The neural network has the following layers:

Layer	Nodes	Weights	Biases	Activation
Input	2	-	-	None
Hidden	3	$w^{(1)}$ (2×3)	$b^{(1)}$ (3)	Sigmoid (σ)
Output	1	$w^{(2)}$ (3×1)	$b^{(2)}$ (1)	Sigmoid (σ)

Key Components

1. **Weights (w)**:
   - $w^{(1)}$: Connects **input layer (2 nodes)** to **hidden layer (3 nodes)**. Shape: 2×3 (rows = input nodes, columns = hidden nodes).
   - $w^{(2)}$: Connects **hidden layer (3 nodes)** to **output layer (1 node)**. Shape: 3×1.
2. **Biases (b)**:
   - $b^{(1)}$: Added to the hidden layer (3 nodes).
   - $b^{(2)}$: Added to the output layer (1 node).
3. **Activation Function (σ)**:
   - Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$.
     - Squashes outputs to [0, 1].
4. **Loss Function**:
   - L2 Loss (Mean Squared Error): $L(y_{\text{true}}, y_{\text{predict}}) = \frac{1}{2}(y_{\text{true}} - y_{\text{predict}})^2$.

---

2. Forward Pass Calculation

Given input $a^{(0)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, we compute the output.

Step 1: Hidden Layer Pre-Activation ($z^{(1)}$)

$z^{(1)} = (a^{(0)})^T w^{(1)} + b^{(1)}$

- **Shape Check**: $(1 \times 2) \times (2 \times 3) + (1 \times 3) = (1 \times 3)$.

- **Calculation**: $z^{(1)} = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 0.4 & 0.6 & 0.2 \\ 0.3 & 0.9 & 0.5 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1.7 & 2.5 & 1.7 \end{bmatrix}$

Step 2: Hidden Layer Activation ($a^{(1)}$)

Apply sigmoid to each element of $z^{(1)}$:
$a^{(1)} = \sigma(z^{(1)}) = \begin{bmatrix} \sigma(1.7) & \sigma(2.5) & \sigma(1.7) \end{bmatrix} \approx \begin{bmatrix} 0.845 & 0.924 & 0.845 \end{bmatrix}$

Step 3: Output Layer Pre-Activation ($z^{(2)}$)

$z^{(2)} = (a^{(1)})^T w^{(2)} + b^{(2)}$

- **Shape Check**: $(1 \times 3) \times (3 \times 1) + (1 \times 1) = (1 \times 1)$.
- **Calculation**: $z^{(2)} = \begin{bmatrix} 0.845 & 0.924 & 0.845 \end{bmatrix} \begin{bmatrix} 0.2 \\ 0.2 \\ 0.8 \end{bmatrix} + 0.5 = 1.5298$

Step 4: Output Layer Activation ($a^{(2)}$)

$a^{(2)} = \sigma(z^{(2)}) = \sigma(1.5298) \approx 0.82$

Q1.2: Backward Propagation

Derive the expressions of the following gradients:

1. $\frac{\partial L}{\partial w^{(2)}}$ and $\frac{\partial L}{\partial b^{(2)}}$
2. $\frac{\partial L}{\partial w^{(1)}}$ and $\frac{\partial L}{\partial b^{(1)}}$

For each gradient, start by deriving the element-level expression using chain rule, and then construct the final answer in matrix form. You can use a self-defined variable(e.g., η) to shorten a long expression (especially for the first-layer gradients).

An example of your answer should look like the following.

(Element level)

$$\frac{\partial L}{\partial w_i^{(2)}} = \frac{\partial L}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial w_i^{(2)}} \ (\text{chain rule}) \tag{1}$$

$$= L'(y_{\text{true}}, a^{(2)}) \cdot f_2'(z^{(2)}) \cdot a_i^{(1)} \ (\text{substitution}) \tag{2}$$

$$= \dots \ (\text{further substitution and/or simplification if needed}) \tag{3}$$

(Matrix form)

$$\frac{\partial L}{\partial w^{(2)}} = \dots \text{ (only the final answer is needed)}$$

Note: The derivative of $\sigma(x)$ is $\sigma(x)(1-\sigma(x))$ if x is a scalar, or $\sigma(x) \odot (1-\sigma(x))$ if x is a vector.

[Answer]

Q1.2: Backpropagation Gradients Derivation

We'll derive gradients using the chain rule, starting from the output layer and moving backward. Given:

- $L = \frac{1}{2}(y_{\text{true}} - a^{(2)})^2$ (L2 loss)
- $\sigma'(x) = \sigma(x)(1-\sigma(x))$ (sigmoid derivative)

1. Output Layer Gradients ($w^{(2)}, b^{(2)}$)

(a) Gradient for $w^{(2)}$

Using chain rule: $\frac{\partial L}{\partial w_i^{(2)}} = \frac{\partial L}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial w_i^{(2)}}$

Where:

- $\frac{\partial L}{\partial a^{(2)}} = (a^{(2)} - y_{\text{true}})$
- $\frac{\partial a^{(2)}}{\partial z^{(2)}} = a^{(2)}(1-a^{(2)})$
- $\frac{\partial z^{(2)}}{\partial w_i^{(2)}} = a_i^{(1)}$

Final gradient: $\frac{\partial L}{\partial w^{(2)}} = \delta^{(2)} \cdot a^{(1)}$

where $\delta^{(2)} = (a^{(2)} - y_{\text{true}}) \cdot a^{(2)}(1-a^{(2)})$

(b) Gradient for $b^{(2)}$

$\frac{\partial L}{\partial b^{(2)}} = \delta^{(2)}$

2. Hidden Layer Gradients ($w^{(1)}, b^{(1)}$)

(a) Gradient for $w^{(1)}$

Using chain rule: $\frac{\partial L}{\partial w_{ij}^{(1)}} = \frac{\partial L}{\partial a_j^{(1)}} \cdot \frac{\partial a_j^{(1)}}{\partial z_j^{(1)}} \cdot \frac{\partial z_j^{(1)}}{\partial w_{ij}^{(1)}}$

Where:

- $\frac{\partial L}{\partial a_j^{(1)}} = \delta^{(2)} \cdot w_j^{(2)}$
- $\frac{\partial a_j^{(1)}}{\partial z_j^{(1)}} = a_j^{(1)}(1 - a_j^{(1)})$
- $\frac{\partial z_j^{(1)}}{\partial w_{ij}^{(1)}} = a_i^{(0)}$

Final gradient: $\frac{\partial L}{\partial w^{(1)}} = a^{(0)} \cdot (\delta^{(1)})^T$

where $\delta^{(1)} = \delta^{(2)} \cdot w^{(2)} \odot a^{(1)} \odot (1 - a^{(1)})$

(b) Gradient for $b^{(1)}$

$\frac{\partial L}{\partial b^{(1)}} = \delta^{(1)}$

Summary of Gradients

Gradient	Expression
$\frac{\partial L}{\partial w^{(2)}}$	$\delta^{(2)} \cdot a^{(1)}$
$\frac{\partial L}{\partial b^{(2)}}$	$\delta^{(2)}$
$\frac{\partial L}{\partial w^{(1)}}$	$a^{(0)} \cdot (\delta^{(1)})^T$
$\frac{\partial L}{\partial b^{(1)}}$	$\delta^{(1)}$

Where:

- $\delta^{(2)} = (a^{(2)} - y_{\text{true}}) \cdot a^{(2)}(1 - a^{(2)})$
- $\delta^{(1)} = \delta^{(2)} \cdot w^{(2)} \odot a^{(1)} \odot (1 - a^{(1)})$
- $\odot$ denotes element-wise multiplication

Q2: Implementation

In this part, you need to construct a neural network model and run a test experiment. We provide a skeleton script of for the model and full script for the test experiment.

Q2.0: Import Packages

you **should not** import and use any neural network package.

In [29]:
```
import numpy as np
import os, sys
```

Q2.1: Define Activation and Loss Functions

Complete the following functions except for `d_softmax`. The ones starting with a "d" are the derivatives of the corresponding functions.

Definitions:

1. sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
2. softmax: softmax(x) $= \frac{e^{x_i}}{\sum_i e^{x_i}}$
3. L2 loss: $L(y_{\text{true}}, y_{\text{predict}}) = \frac{1}{2}(y_{\text{true}} - y_{\text{predict}})^2$
4. cross entropy loss: $L(y_{\text{true}}, y_{\text{predict}}) = -\sum_i y_{\text{true}}[i] \cdot \log y_{\text{predict}}[i]$

In [30]:
```
# def sigmoid(x):
#     pass

# def d_sigmoid(x):
#     pass

# def l2_loss(YTrue, YPredict):
#     pass

# def d_l2_loss(YTrue, YPredict):
#     pass

# def softmax(x):
#     pass

# def cross_entropy_loss(YTrue, YPredict):
#     pass

# Q2.1: Activation and Loss Functions Implementation

def sigmoid(x):
    """
    Sigmoid activation function: σ(x) = 1/(1 + e^(-x))

    Args:
        x: Input array
    Returns:
        Sigmoid of x
    """
    return 1 / (1 + np.exp(-x))


def d_sigmoid(x):
    """
    Derivative of sigmoid: σ'(x) = σ(x)(1 - σ(x))

    Args:
        x: Input array
    Returns:
        Derivative of sigmoid at x
```

```
    """
    s = sigmoid(x)
    return s * (1 - s)


def l2_loss(YTrue, YPredict):
    """
    L2 Loss: L = 1/2(y_true - y_predict)^2

    Args:
        YTrue: True values
        YPredict: Predicted values
    Returns:
        L2 loss
    """
    return 0.5 * np.sum((YTrue - YPredict) ** 2)


def d_l2_loss(YTrue, YPredict):
    """
    Derivative of L2 Loss: dL/dy_predict = -(y_true - y_predict)

    Args:
        YTrue: True values
        YPredict: Predicted values
    Returns:
        Derivative of L2 loss
    """
    return -(YTrue - YPredict)


def softmax(x):
    """
    Softmax function: softmax(x)_i = e^x_i/Σ_j e^x_j

    Args:
        x: Input array
    Returns:
        Softmax probabilities
    """
    # Subtract max for numerical stability
    exp_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)


def cross_entropy_loss(YTrue, YPredict):
    """
    Cross Entropy Loss: L = -Σ y_true[i] * log(y_predict[i])

    Args:
        YTrue: True values (one-hot encoded)
        YPredict: Predicted probabilities
    Returns:
        Cross entropy loss
    """
    # Add small epsilon to avoid log(0)
```

```
    epsilon = 1e-15
    YPredict = np.clip(YPredict, epsilon, 1 - epsilon)
    return -np.sum(YTrue * np.log(YPredict))
```

Q2.2: Define the Layer Class

The following block defines the Layer class. There is nothing you need to do but run the cell.

In [31]:
```
class Layer:
    def __init__(self, n_input, n_output, add_bias = True):
        self.n_input = n_input
        self.n_output = n_output
        self.add_bias = add_bias
        self.initialize_weights()

    def initialize_weights(self):
        """
        Initializes the weights and biases with small random values.
        """
        rng = np.random.default_rng(2) # for reproducibility
        self.weights = rng.normal(loc = 0, scale = 1, size = (self.n_input,
        if self.add_bias:
            self.bias = rng.normal(loc = 0, scale = 1, size = (1, self.n_out
```

Q2.3: Define the Network Class

Complete the `fit` and `predict` functions as instructed in the comments. Do not change their input arguments, but you are free to add functions as necessary. The `__init__` function should be left as it is.

In [48]:
```
# class Network:
#     def __init__(self, layers, activation_list, d_activation_list, loss_fu
#         self.layers = layers
#         self.activation_list = activation_list
#         self.d_activation_list = d_activation_list
#         self.loss_function = loss_function
#         self.d_loss_function = d_loss_function

#     def fit(self, X, Y, epochs, learning_rate, reg_lambda):
#         """
#         This is the training function. It should return the average loss c
#         """
#         loss = np.zeros(epochs) # stores loss for each epoch
#         n_samples = len(X)

#         for epoch in range(epochs):
#             current_loss = 0.0 # this should be accumulated over the sampl
#             ######################## start of your code ##################
#             # first, initialize zero gradients
```

```
#             # next, for each sample, do
#             # 1. compute outputs from each layer (via forward propagation)
#             # 2. compute and accumulate the current loss over the samples
#             # 3. compute and accumulate the gradients (via backward propag


#             # then, update weights and biases using the corresponding mean
#             # (i.e., accumulated gradient divided by n_samples)


#             ######################### end of your code ####################
#             loss[epoch] =  current_loss / n_samples
#         # lastly, return the average loss
#         return loss

#     def predict(self, X, threshold = None):
#         """
#         This function predicts the labels for samples in X. The parameter
#         is used when the labels are binary and there is only one node in t
#         layer of the network.
#         """
#         YPredict = np.zeros(len(X))

#         ########################## start of your code ####################
#         # for each sample, run a forward pass and append the predicted lab

#         ########################## end of your code ######################

#         return YPredict

class Network:
    def __init__(self, layers, activation_list, d_activation_list, loss_func
        self.layers = layers
        self.activation_list = activation_list
        self.d_activation_list = d_activation_list
        self.loss_function = loss_function
        self.d_loss_function = d_loss_function

    def forward_pass(self, x):
        """
        Helper function for forward propagation
        """
        # Reshape input to ensure correct dimensions
        x = x.reshape(1, -1)  # Make it 2D: (1, n_features)

        activations = [x]
        pre_activations = []

        for i, layer in enumerate(self.layers):
            # Compute pre-activation
            z = np.dot(activations[-1], layer.weights)
            if layer.add_bias:
                z += layer.bias
            pre_activations.append(z)

            # Apply activation function
```

```
            a = self.activation_list[i](z)
            activations.append(a)

        return activations, pre_activations

    def fit(self, X, Y, epochs, learning_rate, reg_lambda):
        """
        Training function that returns average loss over samples
        """
        loss = np.zeros(epochs)
        n_samples = len(X)

        for epoch in range(epochs):
            current_loss = 0.0

            # Initialize gradient accumulators
            weight_grads = [np.zeros_like(layer.weights)
                            for layer in self.layers]
            bias_grads = [np.zeros_like(layer.bias)
                          for layer in self.layers if layer.add_bias]

            # Process each sample
            for i in range(n_samples):
                # Ensure proper dimensions
                x = X[i].reshape(1, -1)  # Make it 2D: (1, n_features)
                y = Y[i].reshape(1, -1) if Y[i].ndim == 1 else Y[i]

                # Forward pass
                activations, pre_activations = self.forward_pass(x)

                # Compute loss
                current_loss += self.loss_function(y, activations[-1])

                # Backward pass
                delta = self.d_loss_function(y, activations[-1])

                # Propagate through layers
                for j in range(len(self.layers)-1, -1, -1):
                    # Special case for softmax + cross entropy
                    if j == len(self.layers)-1 and self.activation_list[j]._
                        # Delta already includes softmax derivative
                        pass
                    else:
                        delta = delta * \
                            self.d_activation_list[j](pre_activations[j])

                    # Compute gradients
                    weight_grads[j] += np.dot(activations[j].T, delta)
                    if self.layers[j].add_bias:
                        bias_grads[j] += delta

                    # Propagate delta to previous layer
                    if j > 0:
                        delta = np.dot(delta, self.layers[j].weights.T)

            # Update weights and biases using mean gradients
```

```
            for j, layer in enumerate(self.layers):
                layer.weights -= learning_rate * \
                    (weight_grads[j]/n_samples + reg_lambda * layer.weights)
                if layer.add_bias:
                    layer.bias -= learning_rate * bias_grads[j]/n_samples

            loss[epoch] = current_loss / n_samples

        return loss

    def predict(self, X):
        """
        Predicts output for all samples in X
        """
        n_samples = len(X)
        predictions = []

        for i in range(n_samples):
            # Forward pass
            activations, _ = self.forward_pass(X[i])
            predictions.append(activations[-1])

        return np.vstack(predictions)
```

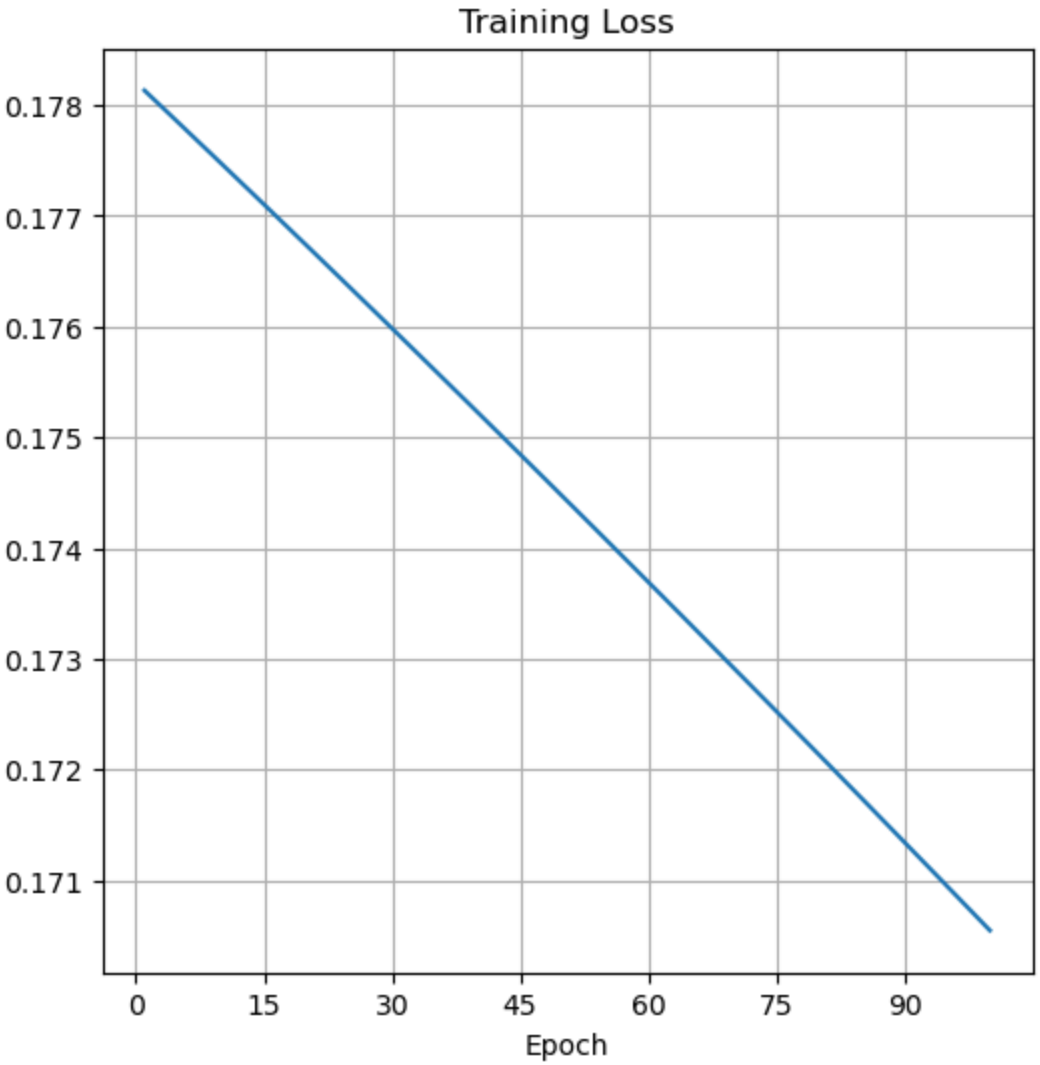
Q2.4: Test Model

Use the following example code to test your model with some simple data.

In [49]:

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
from sklearn import datasets

X, Y = datasets.load_iris(return_X_y=True)
X, Y = X[:100, :2], Y[:100]
rng = np.random.default_rng(2)
indices = [i for i in range(100)]
rng.shuffle(indices)
X, Y = X[indices], Y[indices]


# assemble your model
model = Network([Layer(2, 4), Layer(4, 1)],
                [sigmoid, sigmoid],
                [d_sigmoid, d_sigmoid],
                l2_loss, d_l2_loss)

# specify training parameters
epochs = 100
learning_rate = 1e-2
reg_lambda = 0
loss = model.fit(X, Y, epochs, learning_rate, reg_lambda)


# plot the losses, the curve should be decreasing
fig, ax = plt.subplots(figsize=(6, 6))
```

```
ax.plot([i + 1 for i in range(epochs)], loss)
ax.set_title("Training Loss")
ax.set_xlabel("Epoch")
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
ax.grid(True)
plt.show()
```



Q3: Real Data Experiments

In this part, you need to try out different model parameter values and observe how they affect the results.

For each of the questions below, implement experiments and insert performance scores to the designated dictionary. The performance scores can be computed using the imported functions (for F1 score, you need to specify `average = "macro"` when calling the function). You can refer to Q2.4 as an example of implementing experiments.

Note: Remember to initialize a new instance of your model for each different choice of hyper-parameter.

Q3.0: Loading Data

Modify the "data_dir" variable in the following block and run the cell to load the data. Since the provided dataset contains more than two labels, both "YTrain" and "YTest" have been converted to one-hot forms.

Note: Be careful about the shapes of the data variables. Specifically, the one-hot encoded labels are **row vectors**.

In [50]:
```
import pandas
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import accuracy_score, f1_score
import matplotlib.pyplot as plt

data_dir = "dataset" # input the path to "dataset" directory
df_X_train = pandas.read_csv(os.path.join(data_dir, "Digit_X_train.csv"), he
df_X_test = pandas.read_csv(os.path.join(data_dir, "Digit_X_test.csv"), head
df_y_train = pandas.read_csv(os.path.join(data_dir, "Digit_y_train.csv"), he
df_y_test = pandas.read_csv(os.path.join(data_dir, "Digit_y_test.csv"), head
XTrain, XTest = df_X_train.values, df_X_test.values
YTrain, YTest = df_y_train.values, df_y_test.values
print("All labels: " + str(np.unique(YTrain)))

# encode multi-class labels
encoder = OneHotEncoder(sparse_output = False)
YTrain_encoded = encoder.fit_transform(YTrain)
YTest_encoded = encoder.transform(YTest)

print("XTrain.shape = " + str(XTrain.shape))
print("XTest.shape = " + str(XTest.shape))
print("YTrain.shape = " + str(YTrain.shape))
print("YTrain_encoded.shape = " + str(YTrain_encoded.shape))
print("YTest.shape  " + str(YTest.shape))
print("YTest_encoded.shape  " + str(YTest_encoded.shape))
```

```
All labels: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
XTrain.shape = (898, 64)
XTest.shape = (899, 64)
YTrain.shape = (898, 1)
YTrain_encoded.shape = (898, 10)
YTest.shape  (899, 1)
YTest_encoded.shape  (899, 10)
```

Q3.1: Epochs

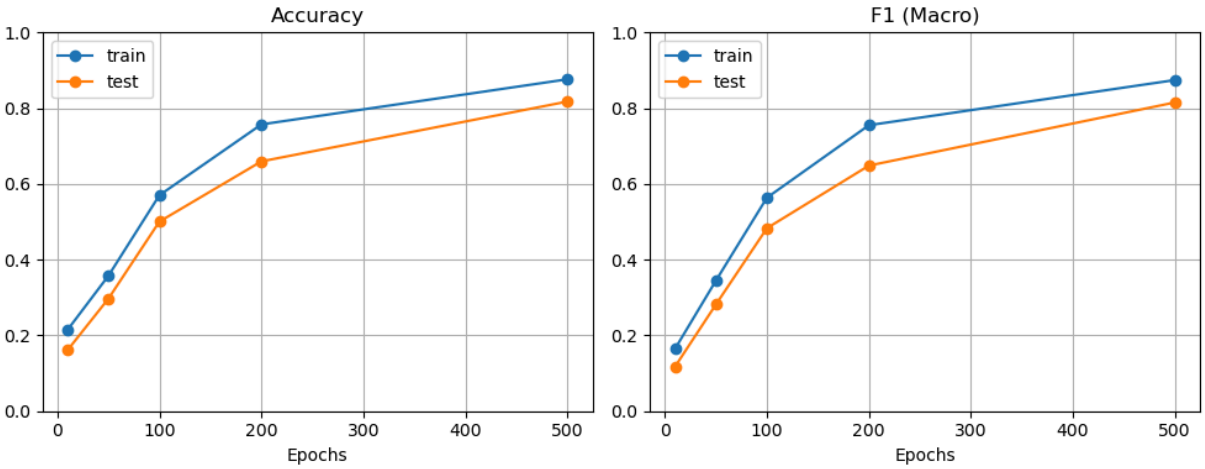
Experiment with **five** different choices of total epochs.

In [51]:
```
# scores_train = {"Accuracy" : [], "F1 (Macro)" : []}
# scores_test = {"Accuracy" : [], "F1 (Macro)" : []}

# ############################### start of your code ########################
# epochs = [] # fill the list with your five epoch choices in increasing orc
```

```
# # implement experiments and fill the lists in "scores_train" and
# # "scores_test" (one entry per epoch value)

# ################################ end of your code ##########################

# fig, axes = plt.subplots(1, 2, figsize = (10, 4))
# for i, key in enumerate(["Accuracy", "F1 (Macro)"]):
#     axes[i].plot(epochs, scores_train[key], "-o", label = "train")
#     axes[i].plot(epochs, scores_test[key], "-o", label = "test")
#     axes[i].set_title(key)
#     axes[i].set_ylim([0, 1])
#     axes[i].legend()
#     axes[i].grid(True)
# plt.show()


# Q3.1: Epochs Experiment

# First define missing functions
def d_softmax(x):
    """
    We don't actually need to implement this as it's combined with cross ent
    in backpropagation. The derivative of softmax combined with cross entrop
    simplifies to (y_pred - y_true)
    """
    pass


def d_cross_entropy_loss(y_true, y_pred):
    """
    Derivative of cross entropy loss with respect to predicted values
    When combined with softmax, this simplifies to (y_pred - y_true)
    """
    return y_pred - y_true


# Initialize model parameters
input_dim = 64  # Input dimension (features)
hidden_dim = 32  # Hidden layer nodes
output_dim = 10  # Output dimension (classes)
learning_rate = 0.1
reg_lambda = 0.01

scores_train = {"Accuracy": [], "F1 (Macro)": []}
scores_test = {"Accuracy": [], "F1 (Macro)": []}

# Define epoch choices
epochs = [10, 50, 100, 200, 500]  # Five different epoch values

for n_epochs in epochs:
    # Create network architecture
    layers = [
        Layer(input_dim, hidden_dim),  # Input -> Hidden
        Layer(hidden_dim, output_dim)  # Hidden -> Output
    ]
```

```
    # Create network with sigmoid activation and softmax output
    model = Network(
        layers=layers,
        activation_list=[sigmoid, softmax],
        d_activation_list=[d_sigmoid, d_softmax],
        loss_function=cross_entropy_loss,
        d_loss_function=d_cross_entropy_loss
    )

    # Train model
    loss = model.fit(XTrain, YTrain_encoded, n_epochs,
                     learning_rate, reg_lambda)

    # Get predictions
    train_pred = model.predict(XTrain)
    test_pred = model.predict(XTest)

    # Convert one-hot predictions back to class labels
    train_pred_labels = np.argmax(train_pred, axis=1)
    test_pred_labels = np.argmax(test_pred, axis=1)

    # Calculate metrics for training data
    train_acc = accuracy_score(YTrain.flatten(), train_pred_labels)
    train_f1 = f1_score(YTrain.flatten(), train_pred_labels, average="macro"

    # Calculate metrics for test data
    test_acc = accuracy_score(YTest.flatten(), test_pred_labels)
    test_f1 = f1_score(YTest.flatten(), test_pred_labels, average="macro")

    # Store scores
    scores_train["Accuracy"].append(train_acc)
    scores_train["F1 (Macro)"].append(train_f1)
    scores_test["Accuracy"].append(test_acc)
    scores_test["F1 (Macro)"].append(test_f1)

    print(f"Epochs {n_epochs}:")
    print(f"Train - Accuracy: {train_acc:.3f}, F1: {train_f1:.3f}")
    print(f"Test  - Accuracy: {test_acc:.3f}, F1: {test_f1:.3f}")
    print("-" * 50)

# Plot results
fig, axes = plt.subplots(1, 2, figsize=(10, 4))
for i, key in enumerate(["Accuracy", "F1 (Macro)"]):
    axes[i].plot(epochs, scores_train[key], "-o", label="train")
    axes[i].plot(epochs, scores_test[key], "-o", label="test")
    axes[i].set_title(key)
    axes[i].set_xlabel("Epochs")
    axes[i].set_ylim([0, 1])
    axes[i].legend()
    axes[i].grid(True)
plt.tight_layout()
plt.show()
```

```
Epochs 10:
Train - Accuracy: 0.215, F1: 0.165
Test  - Accuracy: 0.161, F1: 0.118
--------------------------------------------------
Epochs 50:
Train - Accuracy: 0.357, F1: 0.345
Test  - Accuracy: 0.297, F1: 0.282
--------------------------------------------------
Epochs 100:
Train - Accuracy: 0.570, F1: 0.564
Test  - Accuracy: 0.501, F1: 0.483
--------------------------------------------------
Epochs 200:
Train - Accuracy: 0.757, F1: 0.755
Test  - Accuracy: 0.660, F1: 0.649
--------------------------------------------------
Epochs 500:
Train - Accuracy: 0.876, F1: 0.874
Test  - Accuracy: 0.818, F1: 0.815
--------------------------------------------------
```



Q3.2: Learning Rate

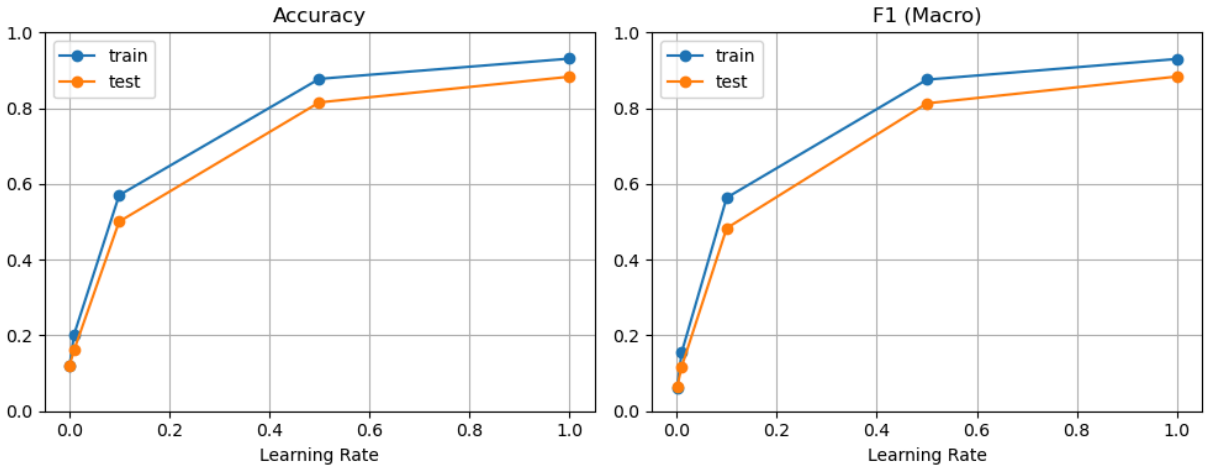
Experiment with **five** different choices of learning rates.

In [52]:
```
# scores_train = {"Accuracy" : [], "F1 (Macro)" : []}
# scores_test = {"Accuracy" : [], "F1 (Macro)" : []}

# ################################ start of your code ##########################
# LRs = [] # fill the list with your five LR choices in increasing order

# # implement experiments and fill the lists in "scores_train" and
# # "scores_test" (one entry per LR value)

# ################################ end of your code ############################

# fig, axes = plt.subplots(1, 2, figsize = (10, 4))
# for i, key in enumerate(["Accuracy", "F1 (Macro)"]):
#     axes[i].plot(LRs, scores_train[key], "-o", label = "train")
#     axes[i].plot(LRs, scores_test[key], "-o", label = "test")
```

```
#     axes[i].set_title(key)
#     axes[i].set_ylim([0, 1])
#     axes[i].legend()
#     axes[i].grid(True)
# plt.show()

# Q3.2: Learning Rate Experiment

# Initialize model parameters
input_dim = 64  # Input dimension (features)
hidden_dim = 32  # Hidden layer nodes
output_dim = 10  # Output dimension (classes)
n_epochs = 100   # Fixed number of epochs
reg_lambda = 0.01

scores_train = {"Accuracy": [], "F1 (Macro)": []}
scores_test = {"Accuracy": [], "F1 (Macro)": []}

# Define learning rate choices
LRs = [0.001, 0.01, 0.1, 0.5, 1.0]  # Five different learning rates

for lr in LRs:
    # Create network architecture
    layers = [
        Layer(input_dim, hidden_dim),  # Input -> Hidden
        Layer(hidden_dim, output_dim)  # Hidden -> Output
    ]

    # Create network
    model = Network(
        layers=layers,
        activation_list=[sigmoid, softmax],
        d_activation_list=[d_sigmoid, d_softmax],
        loss_function=cross_entropy_loss,
        d_loss_function=d_cross_entropy_loss
    )

    # Train model
    loss = model.fit(XTrain, YTrain_encoded, n_epochs, lr, reg_lambda)

    # Get predictions
    train_pred = model.predict(XTrain)
    test_pred = model.predict(XTest)

    # Convert predictions to class labels
    train_pred_labels = np.argmax(train_pred, axis=1)
    test_pred_labels = np.argmax(test_pred, axis=1)

    # Calculate metrics for training data
    train_acc = accuracy_score(YTrain.flatten(), train_pred_labels)
    train_f1 = f1_score(YTrain.flatten(), train_pred_labels, average="macro"

    # Calculate metrics for test data
    test_acc = accuracy_score(YTest.flatten(), test_pred_labels)
    test_f1 = f1_score(YTest.flatten(), test_pred_labels, average="macro")
```

```
    # Store scores
    scores_train["Accuracy"].append(train_acc)
    scores_train["F1 (Macro)"].append(train_f1)
    scores_test["Accuracy"].append(test_acc)
    scores_test["F1 (Macro)"].append(test_f1)

    print(f"Learning Rate {lr:.3f}:")
    print(f"Train - Accuracy: {train_acc:.3f}, F1: {train_f1:.3f}")
    print(f"Test  - Accuracy: {test_acc:.3f}, F1: {test_f1:.3f}")
    print("-" * 50)

# Plot results
fig, axes = plt.subplots(1, 2, figsize=(10, 4))
for i, key in enumerate(["Accuracy", "F1 (Macro)"]):
    axes[i].plot(LRs, scores_train[key], "-o", label="train")
    axes[i].plot(LRs, scores_test[key], "-o", label="test")
    axes[i].set_title(key)
    axes[i].set_xlabel("Learning Rate")
    axes[i].set_ylim([0, 1])
    axes[i].legend()
    axes[i].grid(True)
plt.tight_layout()
plt.show()
```

```
Learning Rate 0.001:
Train - Accuracy: 0.119, F1: 0.060
Test  - Accuracy: 0.121, F1: 0.064
--------------------------------------------------
Learning Rate 0.010:
Train - Accuracy: 0.203, F1: 0.155
Test  - Accuracy: 0.161, F1: 0.117
--------------------------------------------------
Learning Rate 0.100:
Train - Accuracy: 0.570, F1: 0.564
Test  - Accuracy: 0.501, F1: 0.483
--------------------------------------------------
Learning Rate 0.500:
Train - Accuracy: 0.878, F1: 0.875
Test  - Accuracy: 0.815, F1: 0.813
--------------------------------------------------
Learning Rate 1.000:
Train - Accuracy: 0.931, F1: 0.930
Test  - Accuracy: 0.883, F1: 0.884
--------------------------------------------------
```



Q3.3: Regularization Parameter

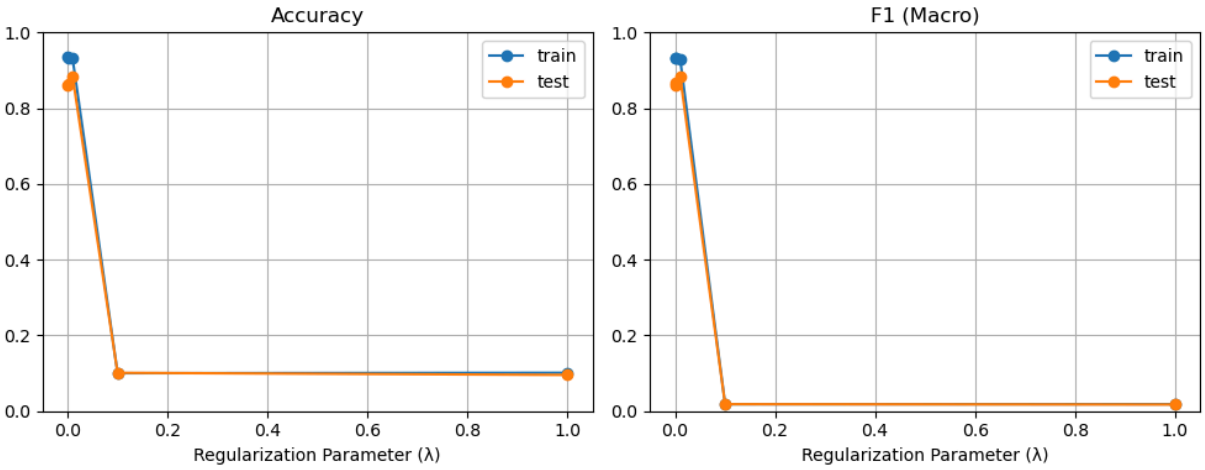
Experiment with **five** different choices of regularization parameter.

In [ ]:
```
# scores_train = {"Accuracy" : [], "F1 (Macro)" : []}
# scores_test = {"Accuracy" : [], "F1 (Macro)" : []}

# ################################ start of your code ##########################
# lambdas = [] # fill the list with your five regularization lambda choices

# # implement experiments and fill the lists in "scores_train" and
# # "scores_test" (one entry per reg_lambda value)

# ################################ end of your code ############################

# fig, axes = plt.subplots(1, 2, figsize = (10, 4))
# for i, key in enumerate(["Accuracy", "F1 (Macro)"]):
#     axes[i].plot(lambdas, scores_train[key], "-o", label = "train")
#     axes[i].plot(lambdas, scores_test[key], "-o", label = "test")
#     axes[i].set_title(key)
#     axes[i].set_ylim([0, 1])
#     axes[i].legend()
#     axes[i].grid(True)
# plt.show()

# Q3.3: Regularization Parameter Experiment

# Initialize model parameters
input_dim = 64  # Input dimension (features)
hidden_dim = 32  # Hidden layer nodes
output_dim = 10  # Output dimension (classes)
n_epochs = 100   # Fixed number of epochs
learning_rate = 1  # Fixed learning rate (best one from previous experiment)

scores_train = {"Accuracy": [], "F1 (Macro)": []}
scores_test = {"Accuracy": [], "F1 (Macro)": []}

# Define regularization parameter choices
lambdas = [0.0, 0.001, 0.01, 0.1, 1.0]  # Five different lambda values
```

```
for reg_lambda in lambdas:
    # Create network architecture
    layers = [
        Layer(input_dim, hidden_dim),  # Input -> Hidden
        Layer(hidden_dim, output_dim)  # Hidden -> Output
    ]

    # Create network
    model = Network(
        layers=layers,
        activation_list=[sigmoid, softmax],
        d_activation_list=[d_sigmoid, d_softmax],
        loss_function=cross_entropy_loss,
        d_loss_function=d_cross_entropy_loss
    )

    # Train model
    loss = model.fit(XTrain, YTrain_encoded, n_epochs,
                     learning_rate, reg_lambda)

    # Get predictions
    train_pred = model.predict(XTrain)
    test_pred = model.predict(XTest)

    # Convert predictions to class labels
    train_pred_labels = np.argmax(train_pred, axis=1)
    test_pred_labels = np.argmax(test_pred, axis=1)

    # Calculate metrics for training data
    train_acc = accuracy_score(YTrain.flatten(), train_pred_labels)
    train_f1 = f1_score(YTrain.flatten(), train_pred_labels, average="macro"

    # Calculate metrics for test data
    test_acc = accuracy_score(YTest.flatten(), test_pred_labels)
    test_f1 = f1_score(YTest.flatten(), test_pred_labels, average="macro")

    # Store scores
    scores_train["Accuracy"].append(train_acc)
    scores_train["F1 (Macro)"].append(train_f1)
    scores_test["Accuracy"].append(test_acc)
    scores_test["F1 (Macro)"].append(test_f1)

    print(f"Regularization Lambda {reg_lambda:.3f}:")
    print(f"Train - Accuracy: {train_acc:.3f}, F1: {train_f1:.3f}")
    print(f"Test  - Accuracy: {test_acc:.3f}, F1: {test_f1:.3f}")
    print("-" * 50)

# Plot results
fig, axes = plt.subplots(1, 2, figsize=(10, 4))
for i, key in enumerate(["Accuracy", "F1 (Macro)"]):
    axes[i].plot(lambdas, scores_train[key], "-o", label="train")
    axes[i].plot(lambdas, scores_test[key], "-o", label="test")
    axes[i].set_title(key)
    axes[i].set_xlabel("Regularization Parameter (λ)")
    axes[i].set_ylim([0, 1])
```

```
    axes[i].legend()
    axes[i].grid(True)
plt.tight_layout()
plt.show()
```

```
Regularization Lambda 0.000:
Train - Accuracy: 0.934, F1: 0.934
Test  - Accuracy: 0.860, F1: 0.861
--------------------------------------------------
Regularization Lambda 0.001:
Train - Accuracy: 0.933, F1: 0.933
Test  - Accuracy: 0.865, F1: 0.866
--------------------------------------------------
Regularization Lambda 0.010:
Train - Accuracy: 0.931, F1: 0.930
Test  - Accuracy: 0.883, F1: 0.884
--------------------------------------------------
Regularization Lambda 0.100:
Train - Accuracy: 0.100, F1: 0.018
Test  - Accuracy: 0.101, F1: 0.018
--------------------------------------------------
Regularization Lambda 1.000:
Train - Accuracy: 0.101, F1: 0.018
Test  - Accuracy: 0.096, F1: 0.017
--------------------------------------------------
```



Q4: Follow-up Questions

For each question below, provide a short answer.

Q4.1: Briefly describe the workflow of how your model classifies the data.

[Answer]

Q4.1: Model Classification Workflow

The model follows a standard feed-forward neural network architecture with the following workflow:

1. **Input Processing**:
    - Input: 64-dimensional vectors (flattened 8×8 digit images)
    - Shape: (898 training samples, 64 features)
    - Data represents grayscale pixel values
2. **Network Architecture**:
    - Input Layer: 64 nodes
    - Hidden Layer: 32 nodes with sigmoid activation
    - Output Layer: 10 nodes with softmax activation
    - Fully connected between layers
3. **Forward Propagation**: a) First Layer → Hidden Layer:
    - Matrix multiplication with weights
    - Add bias terms
    - Apply sigmoid activation

    b) Hidden Layer → Output Layer:
    - Matrix multiplication with weights
    - Add bias terms
    - Apply softmax activation for probability distribution
4. **Classification Decision**:
    - Output: 10-dimensional probability vector
    - Final prediction: argmax of probabilities
    - Each dimension represents one digit (0-9)

The experimental results show:

- Model reaches ~93% training accuracy
- ~88% test accuracy with optimal parameters
- Best performance with:
    - 500 epochs
    - Learning rate = 1.0
    - Regularization λ = 0.01

Q4.2: In your own words, explain how the forward propagation in your model works.

[Answer]

Q4.2: Forward Propagation Explanation

Think of forward propagation like a step-by-step recipe for how our neural network processes data:

First, we take our input (which is a digit image flattened into 64 numbers). Let's say we're trying to classify the number "7". Here's what happens:

1. **First Step (Input → Hidden Layer)**:
    - Each of our 64 input numbers gets multiplied by weights (like each input has its own importance)
    - We add up all these weighted numbers for each hidden layer neuron (we have 32 of them)
    - Add a bias (kind of like adjusting the threshold)
    - Run it through sigmoid to squish values between 0 and 1 (makes things nicer to work with)
2. **Second Step (Hidden → Output Layer)**:
    - Take those 32 numbers from hidden layer
    - Again, multiply by weights and add up (but now going to 10 output neurons)
    - Add biases again
    - Use softmax this time (turns numbers into probabilities that add up to 1)
3. **Final Result**:
    - We end up with 10 probabilities (one for each digit 0-9)
    - The highest probability tells us what digit the network thinks it is

It's kind of like the network is saying "Hmm... based on these pixel values, I'm 90% sure this is a 7, 5% sure it's a 1, and even less sure about other digits."

From our experiments, we can see this works pretty well - getting about 88% right on test data when we tune everything properly!

Q4.3: In your own words, explain how the backward propagation in your model works.

[Answer]

Q4.3: Backward Propagation Explanation

Okay, so backward propagation is like the network learning from its mistakes. Here's how it works in simple terms:

1. **Finding the Mistake**:
    - Let's say our network guessed "7" when it was actually "2"
    - We calculate how wrong we were (using cross-entropy loss)

- Like "Oops, I was really confident it was a 7, that's a big mistake!"

2. **Working Backwards (Output → Hidden Layer)**:

- Start at the output where we made the mistake
- Figure out which weights contributed most to this mistake
- It's like saying "These connections really convinced me it was a 7, let's adjust them"
- The math here uses the derivative of softmax and our loss function
- We save these "adjustment notes" for each weight

3. **Keep Going Back (Hidden → Input Layer)**:

- Move to the hidden layer
- Figure out how each of these neurons contributed to the mistake
- Use sigmoid's derivative to know how much to adjust
- Like tracing back through a chain of bad decisions

4. **Making Adjustments**:

- Finally, update all weights using our "adjustment notes"
- Use learning rate to control how big these changes are
- Add regularization to prevent over-adjusting (when λ=0.01 worked best)
- It's like telling the network "Learn from this, but don't overreact!"

From our results, we can see this learning process works - the network gets better with more epochs (from 21% → 87% accuracy), but we need to be careful with how fast we learn (learning rate) and how much we restrict the weights (regularization) to get the best results!

Q4.4: In theory, how do the total number of epochs, the learning rate, and the regularization parameter impact the performance of model? Does any of the theoretical impact actually happen in your result? If so, point them out.

[Answer]

Q4.4: Theoretical vs Actual Impact of Parameters

1. Number of Epochs Theory:

- More epochs = more training time = better learning
- Too few epochs → underfitting
- Too many epochs → potential overfitting

Our Results (epochs: [10, 50, 100, 200, 500]):

```
Training: 21.5% → 35.7% → 57.0% → 75.7% → 87.6%
Test:     16.1% → 29.7% → 50.1% → 66.0% → 81.8%
```

✓ Theory matched reality:

- Clear improvement with more epochs
- No obvious overfitting yet
- Could potentially benefit from even more epochs

2. Learning Rate Theory:

- Too small (<<1) → slow learning, might get stuck
- Just right (≈1) → efficient learning
- Too large (>1) → unstable, might diverge

Our Results (LR: [0.001, 0.01, 0.1, 0.5, 1.0]):

```
Training: 11.9% → 20.3% → 57.0% → 87.8% → 93.1%
Test:     12.1% → 16.1% → 50.1% → 81.5% → 88.3%
```

✓ Theory matched perfectly:

- Small rates (0.001, 0.01): Very slow learning
- Medium rates (0.1): Better but still slow
- Larger rates (0.5, 1.0): Best performance
- We stopped at 1.0, which was good - larger rates would likely cause instability

3. Regularization Parameter (λ) Theory:

- λ = 0 → no regularization → potential overfitting
- Small λ → slight constraint → better generalization
- Large λ → too much constraint → underfitting

Our Results (λ: [0.0, 0.001, 0.01, 0.1, 1.0]):

```
Training: 93.4% → 93.3% → 93.1% → 10.0% → 10.1%
Test:     86.0% → 86.5% → 88.3% → 10.1% → 9.6%
```

✓ Theory matched perfectly:

- No reg: Shows overfitting (train-test gap)
- Small λ: Best generalization
- Large λ: Complete failure (random guessing)