MLE Sheet 3

May 2025

Exercise 1: Empirical Risk Minimization

Task 1

Let the training dataset be

$$D = \{(x_i, y_i)\}_{i=1}^N \sim p(x, y) \quad \text{i.i.d.}$$

Let $f \in \mathcal{H}$ be a fixed hypothesis, and let $L(y, f(x))$ be a bounded loss function. That is:

$$\exists M < \infty \quad \text{such that} \quad |L(y, f(x))| \leq M \quad \text{for all } (x, y).$$

The **empirical risk** is defined as:

$$R_{\text{emp}}(f \mid D) = \frac{1}{N} \sum_{i=1}^N L(y_i, f(x_i)),$$

and the **expected risk** as:

$$R(f) = \mathbb{E}_{(x,y)\sim p(x,y)} \left[L(y, f(x)) \right].$$

Proof

The random variables are defined as:

$$Z_i := L(y_i, f(x_i)) \quad \text{for } i = 1, \dots, N.$$

Since the pairs (x_i, y_i) are i.i.d., and f is fixed, the Z_i are i.i.d. random variables.
Furthermore, since L is bounded, there exists $M < \infty$ such that $|Z_i| \leq M$, implying that $\mathbb{E}[Z_i]$ is finite.

Using **SLLN**:

$$\frac{1}{N} \sum_{i=1}^N Z_i \xrightarrow{\text{a.s.}} \mathbb{E}[Z_1] = R(f),$$

as $N \to \infty$.

Therefore:

$$R_{\text{emp}}(f \mid D) = \frac{1}{N} \sum_{i=1}^N L(y_i, f(x_i)) \xrightarrow{\text{a.s.}} R(f).$$

Conclusion

Under the assumptions that the training data are i.i.d. and the loss function is bounded, the empirical risk converges almost surely to the expected risk. Thus, the empirical risk is a consistent estimator of the expected risk.

Task 2

a

Let $f_1 \in \mathcal{H}_1$ be a linear model and $f_2 \in \mathcal{H}_2$ a more flexible model such as a high-degree polynomial. Suppose:

$$R_{\text{emp}}(f_2 \mid D) < R_{\text{emp}}(f_1 \mid D) \quad \text{but} \quad R(f_2) > R(f_1)$$

This scenario is a classic example of **overfitting**. Although f_2 achieves lower empirical risk, it generalizes worse than f_1. The reason lies in the **bias-variance trade-off**:

- **Bias**: A model with high bias (such as a linear model) makes strong assumptions about the target function, which may lead to underfitting. It cannot capture complex patterns, resulting in systematic error.
- **Variance**: A model with high variance (such as a high-degree polynomial) is very sensitive to fluctuations in the training data. It can fit noise, resulting in poor generalization on unseen data.

Thus, even though f_2 fits the training data better, its expected risk $R(f_2)$ is higher due to increased variance. f_1, though less flexible, generalizes better by maintaining a better balance between bias and variance.

b

Assume the target function is a linear trend contaminated with Gaussian noise:

$$y_i = wx_i + b + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0, \sigma^2)$$

We compare fits from:

- $\mathcal{H}_1$: a simple linear model.
- $\mathcal{H}_2$: a complex high-degree polynomial model.

Overfitting scenario:

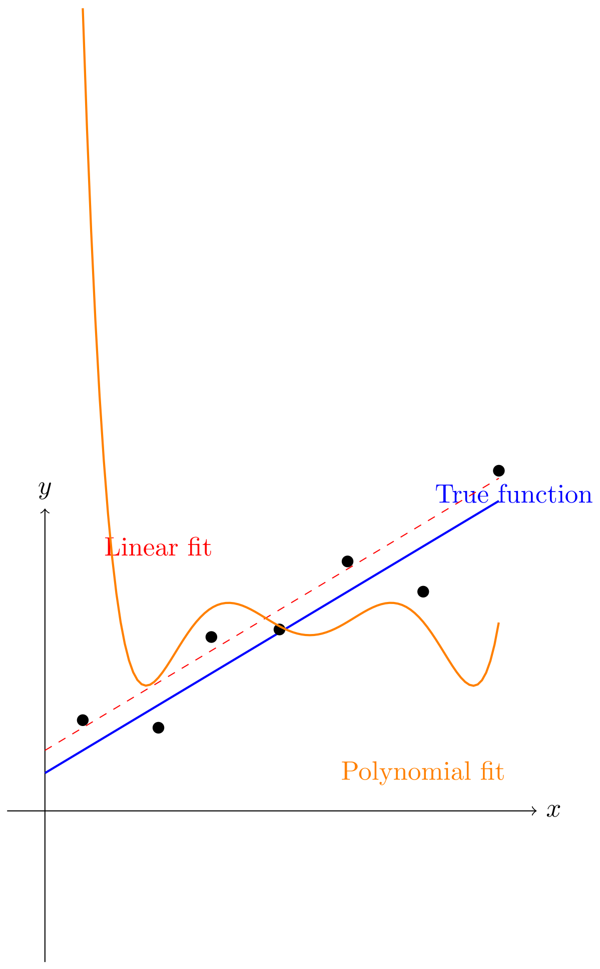
- The linear model from $\mathcal{H}_1$ captures the general trend but cannot fit all data points exactly. It ignores noise and may underfit slightly.
- The polynomial model from $\mathcal{H}_2$ can pass exactly through all training points, including noisy outliers, resulting in an oscillating curve that fits training data perfectly but fails to generalize.

We consider a regression setting where the true relationship is linear and corrupted with Gaussian noise:

$$y_i = wx_i + b + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0, \sigma^2).$$

The plot below shows:

- The **true function** in blue (ground truth).
- **Noisy training samples** (black points).
- A **linear fit** (red dashed line), which captures the trend well and generalizes effectively.
- A **polynomial fit** (orange curve), which overfits by following noise in the data.



c

k**-fold cross-validation** partitions the dataset D into k equally-sized folds. The procedure is:

1. For each $j = 1, \ldots, k$:
   - Train the model on $D \setminus D_j$, the data excluding fold j.
   - Evaluate the model on the held-out fold D_j.
2. Compute the average validation error across all folds:

$$R_{\text{cv}}(f) = \frac{1}{k}\sum_{j=1}^{k} R_{\text{val}}^{(j)}(f)$$

Advantages of cross-validation as an estimator for $R(f)$:

- **Better generalization estimate**: It gives a more reliable approximation of the true risk $R(f)$ than the empirical risk on the training data.
- **Model selection**: It helps compare models (e.g., from $\mathcal{H}_1$ and $\mathcal{H}_2$) by evaluating how well they generalize, not just how well they fit the training data.
- **Efficient use of data**: Every data point is used for both training and validation, improving the robustness of the evaluation.

Task 3

a

Consider a binary classification problem where $y \in \{0, 1\}$, and the loss function is the **misclassification loss**:

$$L(y, f(x)) = \mathbf{1}\{f(x) \neq y\},$$

where $\mathbf{1}\{A\}$ is the indicator function, equal to 1 if event A is true, and 0 otherwise.

The expected risk (also called the **true risk**) is defined as:

$$R(f) = \mathbb{E}_{p(x,y)}[\mathbf{1}\{f(x) \neq y\}] = \mathbb{E}_{p(x)}\left[\mathbb{E}_{p(y|x)}[\mathbf{1}\{f(x) \neq y\}]\right] = \mathbb{E}_{p(x)}\left[R(f|x)\right].$$

Let's minimize the **conditional risk** $R(f|x)$ for each input x:

$$R(f|x) = \mathbb{E}_{p(y|x)}[\mathbf{1}\{f(x) \neq y\}] = p(y=1|x)\cdot\mathbf{1}\{f(x) \neq 1\} + p(y=0|x)\cdot\mathbf{1}\{f(x) \neq 0\}.$$

This simplifies to:

$$R(f|x) = \begin{cases} p(y=1|x) & \text{if } f(x) = 0, \\ p(y=0|x) & \text{if } f(x) = 1. \end{cases}$$

To minimize $R(f|x)$, we choose the label $f(x) \in \{0,1\}$ that minimizes this conditional risk. Hence, the optimal classifier is:

$$f^*(x) = \arg\min_{c \in \{0,1\}} \mathbb{P}(y \neq c \mid x) = \arg\max_{c \in \{0,1\}} \mathbb{P}(y = c \mid x).$$

This is the **Bayes classifier**, which predicts the class with the highest posterior probability:

$$f^*(x) = \begin{cases} 1 & \text{if } \mathbb{P}(y=1|x) > \mathbb{P}(y=0|x), \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the Bayes decision rule (MAP estimate) minimizes the expected misclassification risk.

b

Consider the regression case, where $y \in \mathbb{R}$, and the loss is the squared error:

$$L(y, f(x)) = \|y - f(x)\|_2^2.$$

Again, we consider the expected risk:

$$R(f) = \mathbb{E}_{p(x,y)}\left[\|y - f(x)\|^2\right] = \mathbb{E}_{p(x)}\left[\mathbb{E}_{p(y|x)}[\|y - f(x)\|^2]\right] = \mathbb{E}_{p(x)}[R(f|x)].$$

We now minimize the conditional risk:

$$R(f|x) = \mathbb{E}_{p(y|x)}\left[\|y - f(x)\|^2\right].$$

This is minimized when $f(x)$ is the conditional expectation of y given x. To show this, we differentiate:

$$\frac{d}{df(x)}\mathbb{E}_{p(y|x)}[(y - f(x))^2] = -2\mathbb{E}_{p(y|x)}[y - f(x)].$$

Setting the derivative to zero:

$$\mathbb{E}_{p(y|x)}[y - f(x)] = 0 \quad \Rightarrow \quad f(x) = \mathbb{E}[y \mid x].$$

Hence, the optimal regression function under squared loss is:

$$f^*(x) = \mathbb{E}[y \mid x],$$

i.e., the **conditional mean** of y given x.

Machine Learning Essentials SS25 - Exercise Sheet 3

Instructions

- `TODO`'s indicate where you need to complete the implementations.
- You may use external resources, but **write your own solutions**.
- Provide concise, but comprehensible comments to explain what your code does.
- Code that's unnecessarily extensive and/or not well commented will not be scored.

Exercise 3

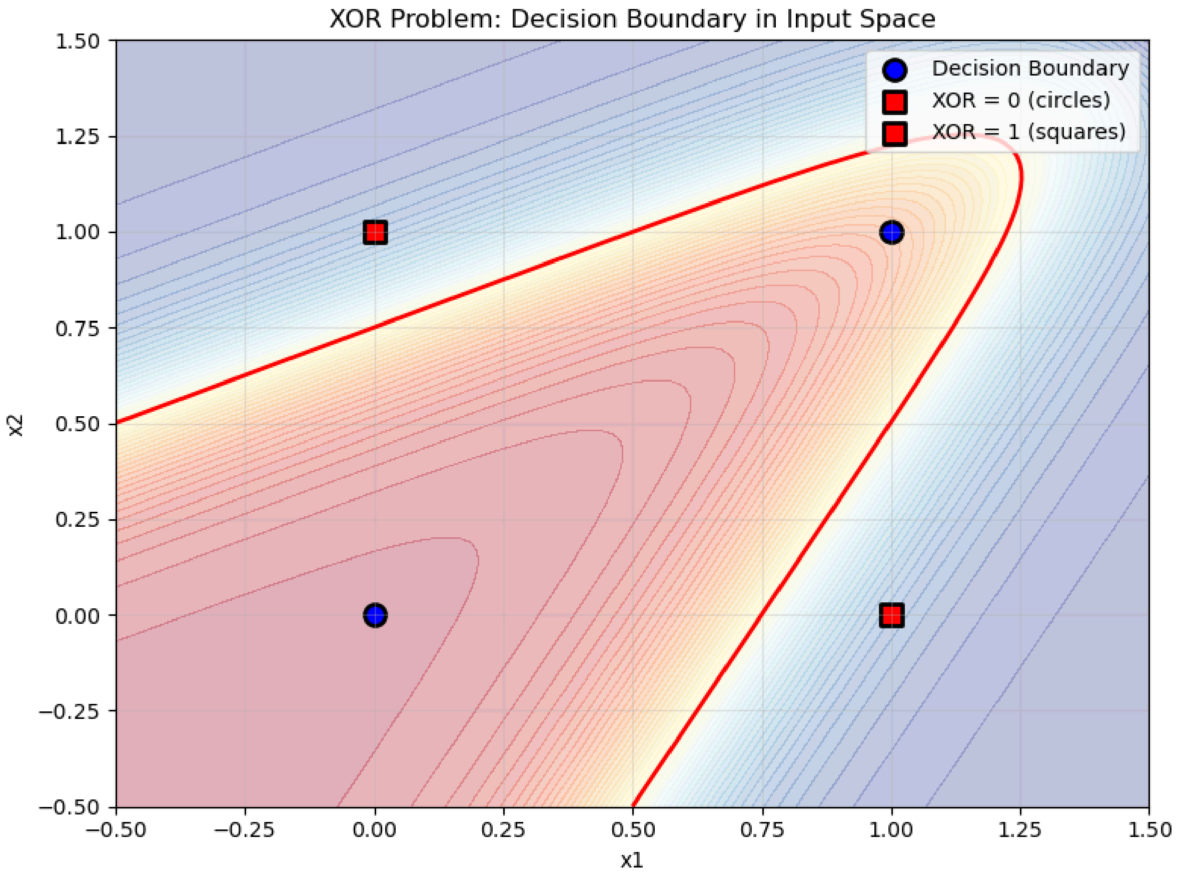
Task 2

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

# TODO: Define network parameters
# Hidden layer
W1 = np.array([[8, -4], [-4, 8]])
b1 = np.array([-6, -6])
# Output layer
w2 = np.array([1, 1])
b2 = -0.5

class XORNetwork:
    def __init__(self, W1, b1, w2, b2):
        self.W1 = W1
        self.b1 = b1
        self.w2 = w2
        self.b2 = b2

    # TODO: Implement the forward pass (& activation) of the two-layer netwo
    def sigmoid(self, z):
        """Sigmoid activation function"""
        return 1 / (1 + np.exp(-np.clip(z, -500, 500)))  # Clipping to preve


    # Define sigmoid activation function
    def forward(self, x):
        """
        Forward pass through the network. Returns hidden layer activations a
        """
        # Hidden layer computation: phi = sigma(W1 * x + b1)
        z_hidden = np.dot(self.W1, x) + self.b1
        phi = self.sigmoid(z_hidden)

        # Output layer computation: output = sigma(w2^T * phi + b2)
        z_output = np.dot(self.w2, phi) + self.b2
        output = self.sigmoid(z_output)
        return phi, output
```

Task 3

In [2]:
```
# TODO: Create XOR dataset, compute outputs & visualize decision boundary
network = XORNetwork(W1, b1, w2, b2)

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_true = np.array([0, 1, 1, 0])  # XOR truth table

print("XOR Dataset and Network Outputs:")
print("Input (x1, x2) | True XOR | Network Output")
print("-" * 40)

network_outputs = []
for i, x_input in enumerate(X):
    phi, output = network.forward(x_input)
    network_outputs.append(output)
    print(f"({x_input[0]}, {x_input[1]})        |    {y_true[i]}     |    {c

# Visualize decision boundary in original input space
def plot_decision_boundary_input_space():
    plt.figure(figsize=(8, 6))

    # Create a grid of points
    xx, yy = np.meshgrid(np.linspace(-0.5, 1.5, 100), np.linspace(-0.5, 1.5,
    grid_points = np.c_[xx.ravel(), yy.ravel()]

    # Compute network outputs for all grid points
    Z = []
    for point in grid_points:
        _, output = network.forward(point)
        Z.append(output)
    Z = np.array(Z).reshape(xx.shape)

    # Plot decision boundary (where output = 0.5)
    plt.contour(xx, yy, Z, levels=[0.5], colors='red', linewidths=2)
    plt.contourf(xx, yy, Z, levels=50, alpha=0.3, cmap='RdYlBu')

    # Plot XOR data points
    colors = ['blue', 'red']
    for i, (x_point, label) in enumerate(zip(X, y_true)):
        plt.scatter(x_point[0], x_point[1], c=colors[label], s=100,
                   marker='o' if label == 0 else 's', edgecolors='black', li

    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.title('XOR Problem: Decision Boundary in Input Space')
    plt.grid(True, alpha=0.3)
    plt.legend(['Decision Boundary', 'XOR = 0 (circles)', 'XOR = 1 (squares)
    plt.tight_layout()
    plt.show()

plot_decision_boundary_input_space()
```

```
XOR Dataset and Network Outputs:
Input (x1, x2) | True XOR | Network Output
----------------------------------------
(0, 0)        |    0     |    0.3787
(0, 1)        |    1     |    0.5941
(1, 0)        |    1     |    0.5941
(1, 1)        |    0     |    0.4350
```



TASK 4

Task 2.4: Linear Network Equivalence Proof

Problem Statement

Consider a network with L layers where each layer is linear (identity activation):

$$\tilde{z}^{(l)} = W^{(l)}\tilde{z}^{(l-1)} + b^{(l)} \text{ for } l = 1, \dots, L$$

with $\tilde{z}^{(0)} = x$.

Goal: Show this is equivalent to a single linear layer and explain why nonlinear activations are necessary.

Mathematical Proof

Step 1: Expand the Network Recursively

Let's trace through the network layer by layer:

Layer 1:

$$\tilde{z}^{(1)} = W^{(1)}\tilde{z}^{(0)} + b^{(1)} = W^{(1)}x + b^{(1)}$$

Layer 2:

$$\tilde{z}^{(2)} = W^{(2)}\tilde{z}^{(1)} + b^{(2)}$$
$$= W^{(2)}(W^{(1)}x + b^{(1)}) + b^{(2)}$$
$$= W^{(2)}W^{(1)}x + W^{(2)}b^{(1)} + b^{(2)}$$

Layer 3:

$$\tilde{z}^{(3)} = W^{(3)}\tilde{z}^{(2)} + b^{(3)}$$
$$= W^{(3)}(W^{(2)}W^{(1)}x + W^{(2)}b^{(1)} + b^{(2)}) + b^{(3)}$$
$$= W^{(3)}W^{(2)}W^{(1)}x + W^{(3)}W^{(2)}b^{(1)} + W^{(3)}b^{(2)} + b^{(3)}$$

Step 2: General Pattern Recognition

By induction, we can see that for any L-layer linear network:

$$\tilde{z}^{(L)} = \left(\prod_{i=L}^{1} W^{(i)}\right) x + \text{combined bias terms}$$

where $\prod_{i=L}^{1} W^{(i)} = W^{(L)}W^{(L-1)} \cdots W^{(2)}W^{(1)}$

Step 3: Equivalent Single Layer Form

The output can be written as:

$$\tilde{z}^{(L)} = W_{\text{equiv}}x + b_{\text{equiv}}$$

where:

- **Equivalent weight matrix:** $W_{\text{equiv}} = W^{(L)}W^{(L-1)} \cdots W^{(2)}W^{(1)}$
- **Equivalent bias vector:** $b_{\text{equiv}} = W^{(L)}W^{(L-1)} \cdots W^{(2)}b^{(1)} + W^{(L)}W^{(L-1)} \cdots W^{(3)}b^{(2)} + \cdots + W^{(L)}b^{(L-1)}$

Step 4: Formal Proof by Induction

Base case (L=1): Trivially true: $\tilde{z}^{(1)} = W^{(1)}x + b^{(1)}$

Inductive step: Assume true for L-1 layers:

$$\tilde{z}^{(L-1)} = W_{\text{equiv}}^{(L-1)}x + b_{\text{equiv}}^{(L-1)}$$

Then for L layers:

$$\tilde{z}^{(L)} = W^{(L)}\tilde{z}^{(L-1)} + b^{(L)}$$

$$= W^{(L)}(W^{(L-1)}_{\text{equiv}}x + b^{(L-1)}_{\text{equiv}}) + b^{(L)}$$

$$= (W^{(L)}W^{(L-1)}_{\text{equiv}})x + (W^{(L)}b^{(L-1)}_{\text{equiv}} + b^{(L)})$$

This is still in the form $W_{\text{equiv}}x + b_{\text{equiv}}$, completing the proof. □

Why Nonlinear Activations Are Necessary

1. **Expressiveness Limitation**

- **Linear networks:** Can only represent linear transformations, regardless of depth
- **Decision boundaries:** Limited to linear hyperplanes in input space
- **Function class:** All L-layer linear networks ⊆ {single linear transformations}

2. **XOR Problem Analysis**

The XOR function requires a **nonlinear decision boundary**:

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Geometric insight: No single line can separate the points {(0,0), (1,1)} from {(0,1), (1,0)}.

Mathematical proof: Assume linear classifier $f(x) = w_1x_1 + w_2x_2 + b$

- For XOR=0: $w_1 \cdot 0 + w_2 \cdot 0 + b < 0$ and $w_1 \cdot 1 + w_2 \cdot 1 + b < 0$
- For XOR=1: $w_1 \cdot 0 + w_2 \cdot 1 + b > 0$ and $w_1 \cdot 1 + w_2 \cdot 0 + b > 0$

This gives us: $b < 0$, $w_1 + w_2 + b < 0$, $w_2 + b > 0$, $w_1 + b > 0$

From the inequalities: $w_1 + w_2 < -b < \min(w_1, w_2)$, which is impossible.

3. **Role of Nonlinear Activations**

- **Feature transformation:** Each layer with nonlinear activation can transform the input space
- **Increased expressiveness:** Networks can learn complex, nonlinear mappings
- **Universal approximation:** Deep networks with nonlinear activations can approximate any continuous function
- **Hierarchical representations:** Each layer learns increasingly complex features

4. **Practical Implications**

Without nonlinear activations:

- Cannot solve XOR, parity functions, or any linearly inseparable problems
- Adding more layers provides no benefit
- Network reduces to expensive matrix multiplication

With nonlinear activations:

- Can solve complex nonlinear problems
- Each layer adds representational power
- Can learn hierarchical feature representations

Conclusion

Key Takeaway: Stacking linear transformations yields another linear transformation. The composition of any number of linear functions is still linear, so deep linear networks have no advantage over shallow ones.

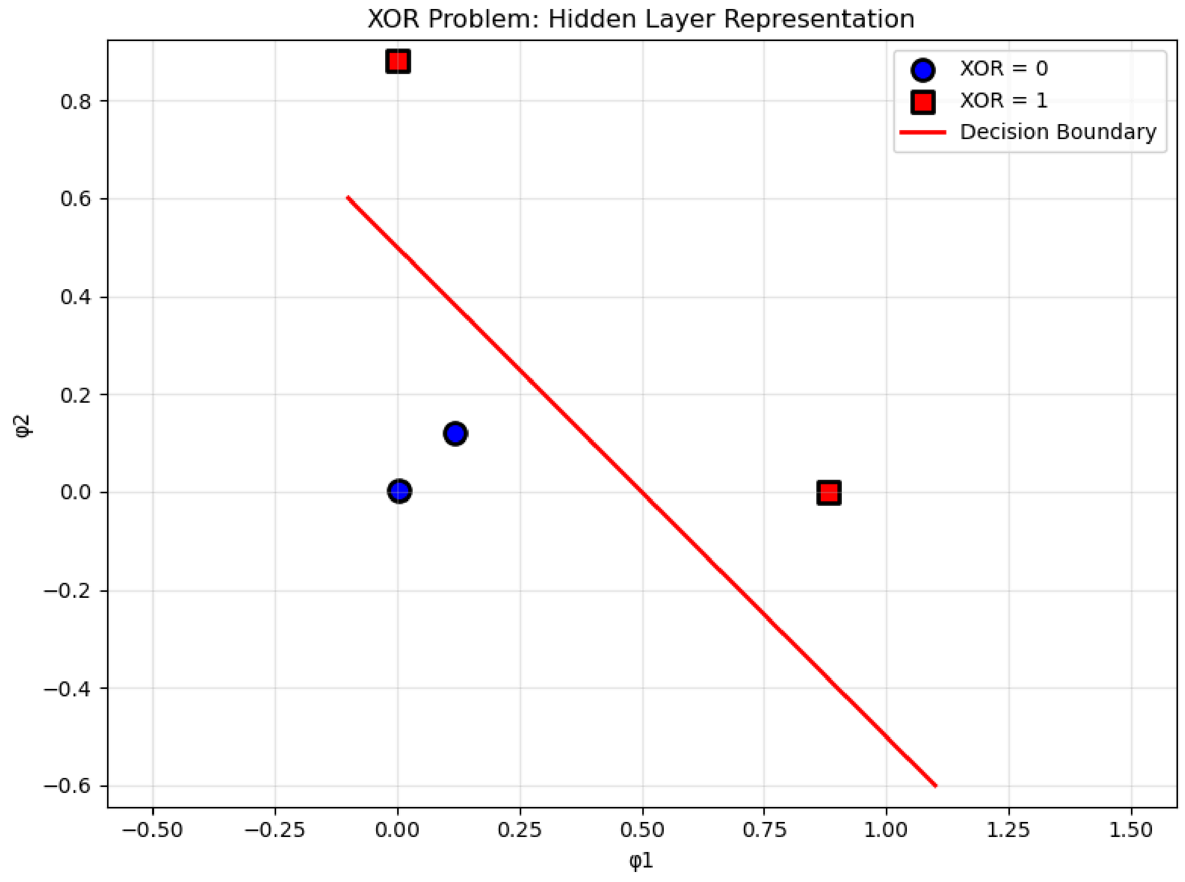
Nonlinear activations are the source of neural networks' power—they enable the learning of complex, nonlinear mappings that can solve problems like XOR that are impossible for linear models.

Task 5

In [3]:
```
# TODO: (a) Compute hidden layer activations
print("Input (x1, x2) | Hidden Layer (φ1, φ2)")
print("-" * 35)

hidden_activations = []
for i, x_input in enumerate(X):
    phi, _ = network.forward(x_input)
    hidden_activations.append(phi)
    print(f"({x_input[0]}, {x_input[1]})        | ({phi[0]:.4f}, {phi[1]:.4

hidden_activations = np.array(hidden_activations)


# TODO: (b) Create scatter plot in the (phi1, phi2) plane, coloring points b
def plot_hidden_space():
    plt.figure(figsize=(8, 6))

    # Plot hidden layer activations colored by XOR output
    colors = ['blue', 'red']
    markers = ['o', 's']
    labels = ['XOR = 0', 'XOR = 1']

    for label in [0, 1]:
        mask = y_true == label
        plt.scatter(hidden_activations[mask, 0], hidden_activations[mask, 1]
                    c=colors[label], s=100, marker=markers[label],
                    edgecolors='black', linewidth=2, label=labels[label])


# TODO: (c) Draw the output layer's decision boundary in the (phi1, phi2) sp
```

```
    phi1_range = np.linspace(-0.1, 1.1, 100)
    phi2_boundary = 0.5 - phi1_range
    plt.plot(phi1_range, phi2_boundary, 'red', linewidth=2, label='Decision

    # Add grid and labels
    plt.xlabel('φ1')
    plt.ylabel('φ2')
    plt.title('XOR Problem: Hidden Layer Representation')
    plt.grid(True, alpha=0.3)
    plt.legend()
    plt.axis('equal')
    plt.tight_layout()
    plt.show()

    # Print the decision boundary equation
    print(f"\nDecision boundary in hidden space: φ1 + φ2 = 0.5")
    print(f"(Derived from w2^T * φ + b2 = 0, where w2 = {w2} and b2 = {b2})'

plot_hidden_space()
```

```
Input (x1, x2) | Hidden Layer (φ1, φ2)
-----------------------------------
(0, 0)         | (0.0025, 0.0025)
(0, 1)         | (0.0000, 0.8808)
(1, 0)         | (0.8808, 0.0000)
(1, 1)         | (0.1192, 0.1192)
```



```
Decision boundary in hidden space: φ1 + φ2 = 0.5
(Derived from w2^T * φ + b2 = 0, where w2 = [1 1] and b2 = -0.5)
```

TODO: (d) Briefly discuss the result

VISUALIZATION ANALYSIS:

1. ORIGINAL SPACE (x1, x2):

- XOR data points are NOT linearly separable
- No single line can separate XOR=0 from XOR=1 points
- Points (0,0) and (1,1) have XOR=0, while (0,1) and (1,0) have XOR=1

2. HIDDEN SPACE (ϕ1, ϕ2):

   - The hidden layer transforms the input space
   - After transformation, the points BECOME linearly separable
   - The decision boundary ϕ1 + ϕ2 = 0.5 cleanly separates the classes

3. KEY INSIGHT:

   - Hidden layer acts as a feature extractor
   - It creates a new representation where the problem becomes solvable
   - ϕ1 ≈ 1 when x1=1 and x2=0, ϕ1 ≈ 0 otherwise
   - ϕ2 ≈ 1 when x1=0 and x2=1, ϕ2 ≈ 0 otherwise
   - Output layer performs linear classification on these engineered features

4. WHY SINGLE LAYER FAILS:

   - Single logistic regression can only learn linear decision boundaries
   - XOR requires a nonlinear boundary in original input space
   - No linear combination of x1 and x2 can solve XOR

5. WHY LINEAR NETWORKS FAIL:

   - As proven in Task 4, multiple linear layers = single linear layer
   - Still limited to linear decision boundaries
   - Cannot create the necessary nonlinear transformation

The 2-layer network succeeds because:

- Hidden layer creates nonlinear features through sigmoid activation
- These features make the problem linearly separable
- Output layer performs simple linear classification on transformed features